FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automated Discovery of Secure Website Domains

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Dominik Frühwirt
Matrikelnummer 0928511

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Dr.techn. Markus Huber MSc
Dipl.-Ing. Dr.techn. Martin Mulazzani

Wien, 15. April 2015

Dominik Frühwirt　　Edgar Weippl

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automated Discovery of Secure Website Domains

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Dominik Frühwirt
Registration Number 0928511

to the Faculty of Informatics
at the Vienna University of Technology

Advisor:	Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance:	Dr.techn. Markus Huber MSc
	Dipl.-Ing. Dr.techn. Martin Mulazzani

Vienna, 15th April, 2015

Dominik Frühwirt   Edgar Weippl

Erklärung zur Verfassung der Arbeit

Dominik Frühwirt
Laimbach 153, 3663 Laimbach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. April 2015

Dominik Frühwirt

Kurzfassung

Durch bekannt gewordene Abhörprogramme, wie das der NSA, rückt die Verschlüsselung von Daten, die über das Internet gesendet werden, immer mehr in den Vordergrund. Viele Webseiten unterstützen mittlerweile das HTTPS Protokoll, das den Traffic zwischen Browser und Webserver mittels TLS absichert. Unglücklicherweise gibt es keine zuverlässige Möglichkeit herauszufinden, ob ein Server HTTPS Verbindungen zulässt. Daher entwickelte die Electronic Frontier Foundation (EFF) die Browser Extension *HTTPS Everywhere*, die das Upgraden einer HTTP Verbindung auf HTTPS automatisiert, wenn es durch den entsprechenden Server unterstützt wird. Dies geschieht durch manuell erstellte und gewartete URL-rewriting Regeln, die mit der Erweiterung mitgeliefert werden. Diese Diplomarbeit befasst sich mit der Problematik der automatisierten Generierung solcher Regeln. Dafür wurde eine Software implementiert, die eine große Anzahl an Domains auf HTTPS Unterstützung prüft und die zugehörigen Regeln erstellt. Die Websites, die über das HTTPS Protokoll erreicht werden können, werden mit den Versionen, die über HTTP erreichbar sind, verglichen, um equivalenten Inhalt und korrekte rewriting-Regeln garantieren zu können. Daher wurden 15 verschiedene Similarity-Matching Methoden implementiert und evaluiert. Das Crawlen der Top Million Websites aus dem Alexa Ranking ermöglichte die Generierung von etwa 190000 einzelner Regeln für fast 129000 verschiedene Domains.

Abstract

Since the large-scale surveillance programs of intelligence agencies like the NSA became known, privacy concerns got in focus of the general public. Many websites support encryption via the HTTPS protocol securing the data transmitted between browsers and webservers by using TLS. Unfortunately, there is no reliable possibility to find out whether a website is available via HTTPS as well. Therefore, the Electronic Frontier Foundation (EFF) developed the browser extension *HTTPS Everywhere* that automates upgrading HTTP connections to secured HTTPS connections if this is supported by the corresponding server. The extension uses manually created and maintained URL-rewriting rules that are shipped with the extension. This diploma thesis investigates the possibilities of automatic rule set generation. For this purpose, a software that checks a large set of domains on HTTPS support and generates the corresponding rules has been implemented. The websites reachable via HTTPS get compared to the versions available via HTTP in order to ensure their equality and correct rewriting rules. Therefore, we implemented and evaluated 15 different similarity matching methods. The large-scale crawl of the Alexa top million websites allowed the generation of about 190,000 single rules for nearly 129,000 different domains.

Contents

CHAPTER 1

Introduction

Since the large-scale surveillance programs of intelligence agencies like the NSA became known, privacy concerns got in focus of the general public. Moreover, the trust in internet companies got tarnished by the fact that governmental organizations eavesdrop even on honest citizens [20].

The *Internet Architecture Board* (IAB), a committee founded for keeping track of the standardization activities of the *Internet Engineering Task Force* (IETF), stated [9] that unencrypted traffic should generally not be sent over the internet anymore. They explain that this could possibly re-establish the trust of users in the internet.

Obviously, this appeal applies to traffic between browsers and web-servers as well. The most popular possibility to access websites in a secure way is to use HTTPS as a communication protocol. The HTTP over TLS protocol provides confidentiality, website authenticity and message integrity [65].

Unfortunately, there is evidence [36] originating from documents recently revealed by Edward Snowden that the NSA is able to decrypt such communications on a large scale. Nevertheless, this does not mean that SSL/TLS is broken at all since the documents also show that decryption is only simple if the intelligence agency got access to the corresponding private key or if TLS is weakly configured [36]. Therefore, HTTPS may still provide protection against surveillance when accessing a website that's key is not compromised. Moreover, other attackers probably do not have vast resources like intelligence agencies and will almost certainly fail on compromising a successfully established TLS session.

Despite the fact that the RFC for HTTPS exists since 2000 [63], HTTPS is not deployed on each and every web-server. This may have several reasons like additional costs for a valid certificate or the effort necessary for configuring the web-server. Moreover, webmasters sometimes are concerned about performance issues caused by the additional computational effort required for encryption. Adam Langley weakened this argument already in 2010 since his measurements showed that enabling TLS requires *"less than 1%*

of the CPU load, less than 10KB of memory per connection and less than 2% of network overhead"[48].

However, even when HTTPS is enabled on a web-server, the insecure HTTP protocol will be also turned on most likely for compatibility reasons since browsers try access via HTTP first [65] when no protocol is explicitly specified in the entered URL. Due to this fact, it may remain undetected that there would be a secure alternative available. Even if a website's administrator enables immediate redirection to the HTTPS equivalent when a user tries to access the website via HTTP, it will still be possible for an active attacker to start attacks like *SSL Stripping* [53] or *Cookie Stealing* [65] against the user.

Even though there are mechanisms like *HTTP Strict Transport Security* (HSTS) or the *secure* tag for cookies that mitigate the risk of such attacks, the user has to rely on the competence of the accessed websites' administrators since these security measures have to be enabled explicitly on server-side.

The browser extension *HTTPS Everywhere* tries to add a further countermeasure against attackers. It ships with a large rule set containing URL rewriting rules for a certain set of websites. The extension instructs the browser to change the protocol from HTTP to HTTPS if a rule exists for the requested website. The entire rule set has been created by the community of *HTTPS Everywhere* and is still maintained by hand. Since the effectivity of the add-on mainly depends on the extent and quality of the rule set it is a desirable goal to enlarge it further.

In this thesis we try to automatize the process of creating new rules in order to achieve this goal. As a first step websites that possibly offer connections via both HTTP and HTTPS have to be found. Since the same website might be accessible on different subdomains when using either HTTP or HTTPS this task might be difficult (e.g. a website may be accessible on *http://example.com/* and on *https://secure.example.com/*). In the following we will call such pairs of URLs comprising one HTTP and one HTTPS-URL *candidates* or *candidate pairs*. Furthermore, administrators may decide to host the actual website on HTTP and the corresponding back-end on the HTTPS port of the same domain. Hence, it has to be checked if the webpages retrieved via HTTP and via HTTPS show the same content. Since webpages are generated dynamically often and may show different content even on two subsequent requests, this is challenging either. Finally, the rule set for the *HTTPS Everywhere* extension has to be created according to the collected data.

As it can be seen, there are three major research questions to answer:

- How is it possible to find potential candidates of HTTP-HTTPS equivalents?
- How can we check two websites regarding equality?
- How can an appropriate rule set for HTTPS Everywhere be created automatically?

The rest of this thesis is organized as follows: In Chapter 2, the technical background of secure websites and browser extensions is introduced. In Chapter 3, related techniques and some similarity matching methods will be explained. Our methodological approach and the design of our software will be presented in Chapters 4 and 5. In Chapter 6 the evaluation procedure will be discussed and in Chapter 7 the results of the large-scale evaluation are presented. Finally, the results are discussed in Chapter 8.

CHAPTER 2

Background

In this section some background knowledge needed for accomplishing the before mentioned goals are explained.

2.1 Transport Layer Security

Transport Layer Security (TLS) can be utilized for authentication and encryption of network traffic. In the ISO OSI-Model it is associated to the presentation layer[65], thus it is located above the transport layer and therefore over TCP and UDP. It builds a transparent provider for encryption to the layers above; hence application layer protocols may work using TLS without the need for adaption. However, for UDP there exists a distinct protocol called *Datagram Transport Layer Security* (DTLS) [64] as TLS is designed for usage with reliable protocols like TCP.

The preceding protocols of TLS are called *Secure Socket Layer* (SSL), but since 1996 the protocol is developed under the name TLS by the IETF. SSL 1.0 has been developed by Netscape Communications in 1994 but has never been released [65]. A few months later, still in 1994, SSL 2.0 replaced its predecessor due to the serious security flaws. In 1996 SSL 3.0 has been released which is also known to be insecure now [58]. The next RFC improving SSL 3.0 has been standardized by the IETF in 1999 and has been renamed to TLS 1.0 in this process. As TLS 1.0 only shows minor changes in comparison to SSL 3.0, the version number used inside of the protocol is 3.1 due to backward compatibility reasons [23] and thus it is also referred to as SSL 3.1. Some further improvements have been introduced by TLS 1.1 [24] in 2006 and TLS 1.2 [25] in 2008. Currently the IETF is working on TLS 1.3 [26] which should mitigate and prevent possible attacks on TLS and also introduce some new features.

TLS makes use of asymmetric cryptography in order to authenticate either one or both communication parties and for exchanging an additional symmetric key. After authentication and key exchange the traffic is encrypted using symmetric cryptography as it costs less than asymmetric encryption regarding computational power [65].

Key Exchange	RSA, Diffie-Hellman
Authentication	RSA, Elliptic Curve Digital Signature Algorithm
Encryption	AES, Triple-DES
Message Authentication	HMAC based on SHA1, MD5

Table 2.1: Examples for Parts of Cipher Suites [26]

In particular, X.509 certificates are utilized to authenticate the communication partner. This standard specifies the format and semantics of the certificates containing the keys necessary for deciding if the counterpart is really the one that should be communicated with.

TLS messages consist of so called records which encapsulate the transmitted data. There are different record types referred to as independent protocols in the RFC [26], namely *handshake*, *alert*, *change cipher spec* and *application data* protocol. In order to make the identification of the encapsulated protocol possible, unique IDs have been assigned to them (e.g. 23 for application data).

Each TLS based connection starts with the TLS *handshake* arranging a common communication base [65]. The communication partners negotiate the TLS/SSL version and the cipher suite which is a combination of a key exchange, an authentication, an encryption and a message authentication method. Some examples for such cipher suites can be seen in Table 2.1.

The *alert protocol* is used for signaling warnings and errors to the communication partner [65]. There are several different warnings like *certificate expired* or *user cancelled*. When one side receives a warning, it is possible to continue the communication. When a fatal error is sent, the connection will be closed immediately after sending the alert [26].

The *change cipher spec protocol* implements only one message saying that every succeeding message will be sent encrypted and authenticated using the negotiated cipher suite [65].

All messages containing application payload are encapsulated using the *application data protocol* [65]. The data may be fragmented and compressed.

Moreover, some TLS configurations are capable of providing forward secrecy [65]. This prevents attackers from decrypting previously recorded TLS data streams even if the long term key got revealed. This is achieved by using distinct session keys for encrypting the communication. These short-term keys are exchanged by using (Elliptic Curve) Diffie-Hellman key exchange over the already long-term key encrypted connection. If a server gets compromised, an attacker will only be able to decrypt currently opened connections but not already closed ones as session keys are deleted after the corresponding connection has ended.

It should be mentioned that TLS can also be configured to only authenticate the traffic but not to encrypt it [26], though this is a hardly used option.

2.2 Hypertext Transfer Protocol (Secure)

HTTP is a stateless application layer protocol which is mainly used to transfer website data. HTTP has been standardized by the IETF and the *World Wide Web Consortium* (W3C) in several RFCs and its latest standardized version is HTTP/1.1 which has been published in 1999 [31]. The first version of HTTP, known as version 0.9, has been released in 1991 [76] and was one of the key technologies for the world wide web. In 1996 RFC 1945 has been finished which defines HTTP/1.0 [7]. HTTP/2 is under development at the time of writing and addresses several performance problems of its predecessors [6].

HTTP usually uses the reliable TCP protocol, typically on port 80 unless specified otherwise. Without using further extensions HTTP transmits data in plain text.

There are two different kinds of HTTP messages, namely request and response messages. They basically consist of three parts, one *start-line*, the *HTTP header* and *message body* [31].

The *start-line* of a request message is called *request line* and consists of the *method*, the *resource identifier* and the *HTTP version.* The *method* defines the operation that should be executed by the server. An assortment of very common method types are:

- **GET**, to retrieve a document from the server,
- **POST**, which can be used for submitting HTML form data to the server and
- **HEAD**, which is similar to GET but does request the server only to send the header omitting the message body of a resource. It may be used for checking if a resource is present at the server or if it has been changed since last access in order to verify the validity of cached documents.

The *resource identifier* denotes the document on which the requested operation should be performed and the *HTTP version* defines the version of the HTTP protocol used.

A response message's *start-line* is called *status line* and usually also comprises three components. At first the *HTTP version* is specified, followed by the *status code* and and a *reason phrase.* The *status code* is represented by a three digit long number indicating the result of the requested operation, whereas the *reason phrase* is a textual representation of the *status code.*

Basically, *status codes* are divided into five classes which are determined by the first digit of the respective code [31]. These categories are named as follows:

- `1xx` - Informational
- `2xx` - Successful
- `3xx` - Redirection
- `4xx` - Client Error
- `5xx` - Server Error

Status Code	**Reason-Phrase**	**Description**
200	OK	Everything went OK
302	Found	The requested resource is available at another location
404	Not Found	The requested resource could not be found on the server
500	Internal Server Error	Some error occurred on the server while processing the request

Table 2.2: HTTP status code examples [31]

Some frequently occurring concrete *status codes* are listed and described in Table 2.2.

In order to transfer some metadata, appropriate header fields can be set within requests and responses. The different header fields are separated by a single new line (carriage return, line feed). The Host-field in the header is the only one which is mandatory [31], as there may be multiple domain names pointing to the same web-server and it has to be able to differentiate which domain is addressed in order to serve different content for different domain names. An empty line marks the end of the header and coincidentally the start of the message body. The message body carries the payload of the corresponding request or response message. In some cases, like in responses to HEAD requests, the message body is empty.

Listing 2.1 shows a very simple HTTP GET request to a Google web-server, indicating the need of the content of *index.html*. It starts with the request line and comprises only one header line and no message body. The response to this request is shown in Listing 2.2 and contains the status line, some header fields and finally the content of the requested *index.html* document.

As already mentioned before, HTTP is a stateless protocol. For keeping the state of a session, an additional mechanism called HTTP Cookie has been defined in RFC 6265 [5]. This allows storing small pieces of data on the client side by setting certain header fields in the response, so it is possible to maintain a user session. A frequent use case is to store a session key in form of a cookie at the users' browser in order to match the correspondig session to a user who is logged in.

```
GET /index.html HTTP/1.1
Host: www.google.at
```

Listing 2.1: Example HTTP Request

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2014 09:51:21 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=f7ad03f76242d1b2:FF=0:TM=141742748...
Set-Cookie: NID=67=mALKnPVUW0G-k9FTUdmSN5T\_IoADyLTOz1...
P3P: CP="This is not a P3P policy! See http://www.goog...
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic,p=0.02
Transfer-Encoding: chunked

<content of index.html omitted>
```

Listing 2.2: Example HTTP Response

2.2.1 HTTPS

As applications got more and more frequently accessed via HTTP and there were no security measures to protect sensitive information like credentials etc., an approach for transmitting HTTP traffic over a secure TLS channel has been developed and described in RFC 2818 [63]. Basically, HTTPS simply encapsulates all HTTP traffic in TLS application data records and uses TCP port 443 by default.

Cookies can also be set within HTTPS sessions. As some websites are available via both HTTP and HTTPS, it is possible that a cookie set via a secure connection is sent over the unsecured one subsequently, e.g. when clicking on a link pointing to an HTTP website. In order to avoid this, it is possible to set the *secure*-attribute [5]. A user agent is allowed to send cookies with this attribute set only via secure connections, thus making session- or other sensitive cookies more secure.

Web-servers sometimes are configured to send a response with HTTP status code 302 when a resource is requested via HTTP, but HTTPS is supported as well. The *Location* header field then contains the URL to the secured webpage and the user agent will navigate accordingly. Since browsers usually prefer to use HTTP when the user does not specify which protocol to use, the first request to the web-server is still sent in plain. Hence, if an attacker manages it to perform a man-in-the-middle (MITM) attack, he will be able to perform different attacks.

One possible attack would be *Cookie Stealing* [65]. Since the attacker is able to eavesdrop and even tamper unencrypted traffic when being the man in the middle, it is possible that cookies sent along with the first unencrypted request are used for hijacking a currently opened session of a web application. In particular, it is not even necessary for the attacker to wait for the user accessing the application he wants to hijack the session of. As soon as one website is accessed via HTTP, the attacker may forge a HTTP

302 redirection response referring to the website he wants to attack, certainly using an HTTP-URL. Since browsers follow redirects from any one website to any other website, this is no problem. Clearly, this attack only works if the before mentioned *secure*-flag was not set by the web application, as cookies will not be sent over the unsecured HTTP connection then.

SSL Stripping [65] enables the attacker to eavesdrop on communications that are assumed to be encrypted, e.g. online banking websites. As already mentioned before, users frequently enter the domain of the website they want to access into the browser without specifying the protocol to use, i.e. *http://* respectively *https://*. On websites with critical content, the web-server will probably respond with a redirect to the HTTPS counterpart of the HTTP website in order to encrypt the transmitted traffic. At this point the man in the middle may intercepts the traffic and acts like a proxy server. All links and redirects to the HTTPS protocol are stripped, which means they are rewritten to HTTP-URLs or omitted in the case of redirects. This way, the user communicates in clear with the attacker, whereas the attacker establishes a secured HTTPS connection to the server. Now the attacker may read or even tamper data that is assumed to be encrypted. Certainly, this might be noticed by the user as the browser does not show the padlock icon indicating an encrypted connection. However, most users are not aware of how to distinguish an encrypted connection from an unencrypted one [69], thus this attack probably takes place undetected.

This attack has been initially introduced at the BlackHat 2009[1] by Moxie Marlinspike [53] who additionally provides the popular tool sslstrip[2] that performs the described attack automatically.

Furthermore, there exist various problems with the *Public Key Infrastructure* (PKI) of the internet [65]. Briefly speaking, the PKI has been introduced to enable user agents to verify the correctness of the certificates supplied by HTTPS web-servers, thus making the communication channel authenticated and encrypted. Therefore, trusted third parties called *Certificate Authorities* (CAs) sign certificates of domains to prove their validity. Browsers are shipped with a set of trusted CAs including their public keys in order to enable the user agent to validate the signatures of signed certificates.

In [65] some problems of this architecture are described. The first and said to be biggest problem is that any CA is able to issue a certificate for any domain. Therefore, attackers that are able to obtain a certificate for an already registered domain may perform large-scale man-in-the-middle attacks with valid certificates. Since the attacker's certificate is also signed by a valid CA, no warnings will be shown at the user agent and the security indicating padlock of the browser will also be displayed. Furthermore, there is no guarantee that governments are not able to force CAs to issue certificates for global surveillance purposes. In 2011 such a security breach of a CA happened [34]. Attackers managed it to generate over 500 fraudulent certificates including one wildcard certificate for Google (*.google.com). As the security report shows, this certificate has been used for

[1]`https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html`, retrieved: 2015-02-10

[2]`http://www.thoughtcrime.org/software/sslstrip/`, retrieved: 2015-02-10

performing a large-scale man-in-the-middle attack mostly affecting users located in Iran.

However, the consequences of such breaches could be possibly mitigated by reducing the set of trusted CAs. The evaluation presented in [61] shows that 34% of the CAs that are included in common trust stores may be removed without any consequences. The authors analyzed the traffic of their university's network for two months and observed that not a single warning would have been shown by a user agent if the before mentioned set of CAs would have been removed from the trust stores.

A group of renowned organizations including, amongst others, Mozilla, the Electronic Frontier Foundation (EFF) and Cisco Systems, started a project in 2014 called *Let's Encrypt*[3] in order to reduce the effort of obtaining a valid certificate and deploying it on a web-server. They develop a tool that is able to automatize the required steps and furthermore keeps track of the expiration and renewal of an already deployed certificate. Therefore, a dedicated CA will be provided and the process of proving the domain ownership will be automated. *Let's Encrypt* is expected to be released in mid-2015 and aims for a wider deployment of HTTPS in the internet due to the simplicity of the tool and not least because it is free of charge.

2.2.2 HTML

A website comprises at least one Hypertext Markup Language (HTML) file in the simplest case. It is a language used for structuring documents in a semantic way. The development of the most current version, HTML5 [78], has been finished in October 2014 and should redeem its predecessor HTML 4 [77] amongst other languages like XHTML. The HTML specification is maintained by the W3C.

An HTML document structures the contained data by enclosing it with HTML tags which are written in angle brackets. A very basic example for an HTML document can be seen in Listing 2.3. Every document starts with the `html`-tag and ends with the corresponding end-tag. These are denoted by putting a `/` after the opening angle bracket, e.g. `</html>` for the `html`-end-tag. However, there are also tags which can have an optional end-tag and some that must not have one. Additionally, tags may have attributes describing additional information, e.g. the `src`-attribute in the `img`-tag in the given example which specifies the image's location.

Usually, HTML websites additionally include *Cascading Style Sheets* (CSS) for defining the look of the page and *JavaScript* (JS) for executing client side code, e.g. for asynchronous communication and dynamic reloading of content.

[3] `https://letsencrypt.org/`, retrieved: 2015-03-05

```
<html>
  <head>
    <title>Example Document</title>
  </head>
  <body>
    This is an example HTML document. <img src="example.png">
  </body>
</html>
```

Listing 2.3: Example HTML document

2.3 Browser Extensions

In [1], a browser extensions is defined as *"software that optionally adds or removes functionality to the browser"*[1]. Extensions are written by using APIs provided by the browser the extension should be embedded in. Furthermore, there are also components which add functionality but are denoted as browser plugins. The difference is that plugins run partly independent from the browser, thus using external code. Popular examples for plugins are *Oracle Java Plugin* which allows the execution of Java-Applets and *Adobe Flash Player* that enables the browser to display Flash-content.

One concrete extension that occupies a central position in this thesis is named *HTTPS Everywhere*[4]. It evolved from a cooperation of the EFF and *The Tor Project*[5] and is available for multiple popular browsers. In principle, this extension rewrites HTTP-URLs according to a certain rule set in order to upgrade them to HTTPS. These rules are defined in XML-files and make use of regular expressions to accomplish flexible URL rewriting. As even the first request to a web-server is sent over HTTPS when using the *HTTPS Everywhere* extension, *SSL Stripping* attacks are not possible anymore. Moreover, session hijacking by stealing unsecured cookies sent along with the first HTTP request is also prevented. Certainly, this mechanism only works if the accessed website is present in the rule set. However, the rule set is created and maintained by the community of the extension, thus fameless websites are possibly missing. The structure of XML rule files will be explained in Section 4.3.

Some other extensions that are relevant for this thesis are explained in the following.

The extension *Perspectives*[6] [81], available for Firefox Browser, pursues the strategy to request the sight of globally distributed nodes to a certain certificate of a server. When the browser retrieves the public key of a web-server, that is not stored in its cache, i.e. it is visited for the first time, or the cached certificate differs from the one it received, notary nodes are requested to send their knowledge about the certificate of the server. This way, the client should receive the same certificate from all requested notary nodes and can check for attacks that target to eavesdrop or tamper the data sent. Moreover,

[4] `https://www.eff.org/HTTPS-EVERYWHERE`, retrieved: 2015-02-09
[5] `https://www.torproject.org/`, retrieved: 2015-02-09
[6] `http://perspectives-project.org/`, retrieved: 2015-02-09

this approach makes it even possible to validate untrusted, unsigned certificates as an attack to the majority of distributed notary nodes is very unlikely to happen. Hence, a web-server's certificate is assumed to be valid if the fingerprint of the certificate is confirmed by a certain amount of notary nodes.

Moreover, Moxie Marlinspike proposed *Convergence*[7] that bases on the idea of *Perspectives* and has been introduced at his talk at the DEFCON [54] in 2011. A corresponding Firefox extension and the code for running a notary node are available for download.

Certificate Patrol[8] tries to address a similar problem by expanding Firefox' and SeaMonkey's features by certificate pinning. The extension caches (pins) certificates of websites which have been visited before, i.e. they utilize a so called *Trust-On-First-Use* (TOFU) policy. If the certificate changes on future requests, *Certificate Patrol* will show an intrusive warning to the user. However, users get also warned when a website legitimately changed its certificate, e.g. when the old one expired or the private key leaked and a new one was generated, and will probably get desensitized to these messages over time [70].

The extension *Certlock* presented in [70] tries to address this problem by omitting some warnings. They base the decision of when to warn a user on the certificate authorities' (CAs) countries. If the changed certificate origins from a CA of the same country as the old one, no warning will be raised. Unfortunately, we could not find a finished implementation ready to download, but a Google Code page[9] comprising one commit made in 2010 commented with *"not a complete implementation"*. Hence, it is supposed that this extension is not being developed further.

Moreover, there are preloaded key pinning lists for Firefox (starting with version 32 [19]) and for Chrome (since version 13 [49]) for some chosen domains.

The Google Chrome extension *KB SSL Enforcer*[10] is capable of redirecting the user to HTTPS websites without maintaining a list like *HTTPS Everywhere*. When a request is sent via HTTP, it probes if the requested document is available via HTTPS on the same domain and path like the actually requested HTTP resource. If so, it is assumed that this domain supports TLS secured transmission and future requests to the same domain will be sent solely via HTTPS [30]. However, sometimes the extension bricks websites by redirecting the browser to broken or empty websites [43], as the content is not compared in order to assure equality of the HTTP and HTTPS website. Moreover, the extension can not hinder the browser to sends the first request to a website in plain enabling an attacker to steal cookies. Furthermore, it is easily conceivable that an attacker blocks the probing-requests to convince *KB SSL Enforcer* that there is no HTTPS web-server, hence there will be no redirect triggered.

HTTPS Finder[11] represents a very similar extension for Firefox. It exhibits the same weaknesses as *KB SSL Enforcer*, but provides an additional feature for creating rules for

[7]http://convergence.io/

[8]`http://patrol.psyced.org/`, retrieved: 2015-02-09

[9]`https://code.google.com/p/certlock/`, retrieved: 2015-02-09

[10]`https://code.google.com/p/kbsslenforcer/`, retrieved: 2015-02-10

[11]`https://code.google.com/p/https-finder/`, retrieved: 2015-02-10

HTTPS Everywhere out of gathered information. Additionally, on the homepage of this extension there is a recommendation to create and deploy such rule files whenever possible, as *HTTPS Everywhere* improves the security more than *HTTPS Finder*, probably because of the before mentioned problems.

All of the before mentioned approaches are able to prevent man-in-the-middle and *SSL Stripping* attacks under certain conditions. Nevertheless, most of them are vulnerable to attacks that take place when a website is visited for the very first time.

CHAPTER 3

State of the Art

Ristić [65] describes several different problems with HTTPS. One of them is that there is no possibility for a user agent to determine if a website supports TLS secured communication, thus it simply uses an unencrypted connection, i.e. HTTP, when no protocol is specified explicitly. Furthermore, it is stated that browsers are very tolerant in terms of certificate problems. Users can skip certificate warnings issued by the user agent without any problems and most of them do so as various studies show [72, 22]. This way, active attackers may utilize self-signed certificates to perform man-in-the-middle attacks which indeed cause the user agent to display a warning, but still be effective due to the unawareness of users. There is another issue with mixing contents from both encrypted and unencrypted sources. HTTPS websites sometimes embed content (e.g. images) that reside on a server that does not support TLS connections. User agents tolerate this to a certain extent, thus an active attacker is able to modify these resources and compromise the session. Cookies can be, as already described in Section 2.2.1, protected from active attackers by setting the *secure* flag. However, most of the web applications do not set this attribute [46]. Therefore, it will still be possible to steal cookies or perform *SSL Stripping* if the user agent sends a single request to the web-server over the unsecured HTTP protocol.

One method to mitigate all of the before mentioned attacks is *HTTP Strict Transport Security* (HSTS). It is defined in RFC 6797 [41] which was released in 2012 and describes a mechanism deployed at both the web-server and the user agent. Websites that are reachable over HTTPS may instruct the browser to communicate with this server exclusively over secured connections in the future and to treat every certificate issue as a fatal error, i.e. the user must have no possibility to skip a certificate warning on the protected domain. This is achieved by setting the *Strict-Transport-Security* header field in an HTTP response sent over a secured connection, e.g. TLS. User agents react to the header by sending no further requests over unsecured connections to the domain it received the HSTS header from. Furthermore, there is one mandatory and one optional directive to parameterize HSTS' behavior. The mandatory *max-age* directive specifies the number of seconds the user agent should remember the HSTS setting for the corresponding domain. The browser will update the expiry date every time the user visits the website. If all subdomains of a certain domain also support HTTPS connections it is possible to set the *includeSubDomains* directive that instructs the user agent to apply the HSTS policy to subdomains as well. In addition to the possibility of dynamically setting the HSTS policy for websites by using the *Strict-Transport-Security* header field, some browsers come with preloaded HSTS databases for certain popular domains [46].

Unfortunately, HSTS is not a comprehensive protection against active attackers as the first request to a website may still be sent unencrypted [46]. Furthermore, the first request sent after the expiration of the *max-age* time may be transmitted in clear as well, enabling *SSL Stripping* and *Cookie Stealing* again. In addition to these weaknesses, the deployment of HSTS in real environments is, according to the evaluation in [46], is erroneous in nearly 30% of the cases.

For this reason users cannot rely on the correct and universal deployment of HSTS and possibly want to have an additional safety net. As already described in Section 2.3, *HTTPS Everywhere* provides such supplementary security measures. Since the extension's effectiveness mainly depends on the deployed rule set, a more comprehensive set of rules would cover a larger amount of websites, thus leading to enhanced security when browsing the web.

In our approach that aims at generating such rules automatically, it is necessary to have algorithms in place that are able to detect if two retrieved webpages are equal or at least very similar in order to decide if we found an HTTP-HTTPS equivalent.

Such website comparisons have also been utilized for checking the Tor [27] network on compromising nodes. Tor is an anonymization tool that relies on wide spread participating nodes and some exit nodes which are also operated by the community. At these exit nodes it is possible to alter unencrypted traffic of Tor-users, thus performing a kind of man-in-the-middle attack is feasible. In order to identify malicious exit nodes, a tool called *TorFlow* [62] has been developed for comparing websites loaded via Tor with the same website loaded normally. They strip the content only to contain tags which they are interested in, i.e. entities which possibly contain malicious code. Therefore, the website's content, in means of non-source code text or images, is not considered due to the stripping.

Similarity detection is also utilized for finding potential phishing websites. In [44], currently used techniques of phishing webpage detection are summarized. However, not all of them can be used for our purposes as several aim at user training and other non-website-comparing mechanisms. Some of them try to match the source code and/or text of the real page and the potential fishing website. Matching the content (HTML code and text) of phishing pages has shown to be very effective [18]. However, phishers responded to these approaches by putting pages together which solely consist of images, thus showing only `img`-tags to the anti-phishing tools and therefore fooling them. Hence, there are new approaches based on comparing the visual appearance of websites like described in several papers [57, 83, 18, 38].

Moreover, there are multiple approaches that compare documents on source code level, often regarding the hierarchical structure and/or the semantic content. Buttler summarized some of them in [16] and also proposed a new method using shingles which were initially introduced by Broder [14]. In [39], Henzinger provides a large-scale evaluation of different algorithms for finding near-duplicate web pages. Theobald et al. [74] propose an approach which is designed to be mostly sensitive for natural language. Hence, they define near-duplicate web pages as documents showing the same content disregarding framing and other visual components.

Furthermore, it is possible to utilize some approaches [50, 32] designed for XML similarity-matching for comparing the source code of websites since HTML webpages exhibit a hierarchical structure similar to XML documents.

Another method for detecting similar documents is provided by different fuzzy hashing techniques [11, 45, 12, 13, 67, 10]. As opposed to cryptographic hashes which change to a large extent even if only one bit of the input gets flipped, fuzzy hashes should be resistant against small modifications. In this way, it is possible to match slightly modified files, thus finding near duplicates.

A concrete explanation of the before mentioned approaches for similarity detection will be given in the next sections.

3.1 Visual Similarity

Since humans mainly differentiate webpages by taking a look at them rather than reading the source code there are also approaches that try to recognize visual differences algorithmically. A very common application for visual website comparison is phishing webpage detection. In the following some visual similarity approaches for websites are presented.

In [73], the screenshot of a webpage is segmented into several parts which either contain images, textual content or mixed content. After labeling the different sections accordingly, graph matching [40] is applied in order to calculate a similarity value. It should be noted that also pages with related layouts are considered to be similar. In an example given by the authors, screenshots of two different search engines give a positive match.

The method proposed by Fu et al. [35] tries to detect phishing pages by measuring visual similarities based on the *Earth Mover's Distance* (EMD). It is a very natural distance measure which is particularly suitable for *"cognitive distance evaluation"*[35]. They resize the screenshot images to 100x100 pixels and reduce the color space to get a robust and normalized signature. The evaluation shows very high precision and recall values for the proposed algorithm.

The combined approach utilized in [57] tries to match textual as well as visual content. It considers metadata of text blocks like background and foreground color, font size and family and also width and height of embedded images, their histograms and their *2D Haar wavelet transformations* [71]. Moreover, it takes the overall screenshot of the webpage into account, also by extracting histogram and wavelet transformation. A training set has been utilized to find an optimal threshold, such that false negative rate and false positive rate is minimized. The evaluation shows that only two of 42 phishing pages could not be detected to be similar. The authors state that these two misclassified phishing pages do not copy their originals very well.

Another method to compare webpages is to use discriminative keypoint features, as proposed in [18]. The authors use a modification of *Contrast Context Histogram* (CCH) [42] descriptors, called *Lightweight CCH* (L-CCH) to generate the signature of a website. The original CCH has been adapted as its capability to match images scale- and rotation-invariant is not required for webpage comparison. The L-CCH descriptors are gathered around keypoints which are found by applying the Harris-Laplace detector. This method achieves an accuracy of about 95% and false positive and false negative rates are lower than 1%.

The approach proposed in [38] uses the existing tool *imgSeek*[1] for phishing page detection which utilizes wavelet transformation (similar to [57]) for finding similar images inside of a database. It takes an image as input and searches for similar ones. Moreover, it calculates a similarity value for possible similar candidates. By only using the output of *imgSeek* as an indicator for phishing page classification, the false positive rate is at 18%.

Another combined method of visual and textual similarity matching is presented by Zhang et al. in [83]. They utilize the EMD for recognizing visual similarity, but unlike in [35] the threshold is determined by using a Bayesian model. The comparison between their and Fu's image classifier shows that their approach exhibits higher accuracy. Furthermore, they combine textual and visual comparison by using their fusion algorithm which outperforms both of the individual approaches.

The theoretical work of Maurer et al. [55] also focuses on detecting phishing pages by comparing their visual appearances. The authors propose to use the Java library *LIRe*[2] [52] to generate signatures of the webpage screenshots. *LIRe* is a content based image retrieval library that comprises various methods for retrieving similar images. Since there is no finished implementation, no evaluation is given.

3.2 Code Similarity

As already mentioned before, websites are written in the hierarchically structured language HTML. Some similarity algorithms solely rely on the comparison of the structure disregarding the content of the website, others consider both textual content and structural elements of the code. It is also possible to measure resemblance of documents by just taking structural elements into account as the hierarchical construction of an HTML file may suffices for recognizing similar webpages.

Buttler summarizes some document structure similarity algorithms in [16], e.g. the well known *Tree Edit Distance* (TED). This distance is calculated by counting the minimum amount of deletions, insertions and updates that are required to transform one tree into another. As HTML is a hierarchical format with one single root node

[1]`http://sourceforge.net/projects/imgseek/`, retrieved: 2015-02-24
[2]`http://www.semanticmetadata.net/lire/`, retrieved: 2015-02-24

(the `html`-tag), it is possible to convert it into a tree and calculate the TED of two different documents. The similarity is computed by dividing the distance value of the two documents by the maximum possible TED value which is equal to the amount of nodes comprised by the bigger tree. The *Robust Tree Edit Distance* (RTED) [60] is one of multiple algorithms proposed in literature to compute the TED in a runtime efficient way compared to other methods. However, RTED still has a run time complexity of $\mathcal{O}(n^3)$ and a space complexity of $\mathcal{O}(n * m)$, where m and n are the number of elements of the trees to compare.

Moreover, Buttler [16] describes *Weighted Tag Similarity*, one of the simplest similarity measures for structured documents. This method relies on the amount of occurrences of different tags inside of both documents disregarding their ordering. However, it is stated that this method would have very low accuracy for HTML documents due to the limited set of different tags.

Furthermore, an application of *Fast Fourier Transformation* (FFT) for comparing the similarity of two documents is described in [33] and evaluated by Buttler. They transform the sequence of tags comprised by the document into a sequence of numbers, s.t. each distinct tag is converted to the same number in both transformations. These numerical series are treated as time series in order to make the application of FFT possible.

Finally, Buttler [16] proposes a novel approach which utilizes *Shingling* [14] and *Path Similarity*. *Path similarity* simply traverses the structural tree of the HTML document and stores every path contained by the tree. The paths for the HTML example given in Section 2.2 would look like shown in Listing 3.1. The similarity can be calculated by dividing the amount of paths occurring in both documents by the number of paths of the document comprising the most paths. When applying *Shingling*, contiguous subsequences of these paths get hashed and are used for comparison. Therefore, an arbitrarily long sliding window is utilized to generate the hashes of the sub-paths. After that, the sets containing the hashes of both documents are compared and the similarity value is given by the percentage of hashes contained in both sets.

```
/html
/html/head
/html/head/title
/html/head/title/[text]
/html/body
/html/body/[text]
/html/body/[text]/img
```

Listing 3.1: Example Paths

The experiments of Buttler show that the TED algorithm is several orders slower than any of the other algorithms under test. Though, it is assumed that it is the best measure for similarity because the algorithm computes the optimum edit distance provably. Moreover, it is stated that the FFT algorithm is much slower than Weighted Tag or *Path Shingle* algorithm, but still less accurate. The *Weighted Tag Similarity* represents one of the fastest and most accurate algorithms tested in [16].

In [4], Augsten et al. propose the usage of *pq*-Grams for approximating the tree edit distance. A *pq*-Gram is a subtree of the original tree that consists of one node of the original tree, $p-1$ of its ancestors and q of its children. After computing all pq-Grams of a pair of trees, the similarity is calculated using the amount of *pq*-Grams the two trees have in common. The sensitivity of this algorithm can be controlled by adjusting the parameters p and q. The pq-gram distance for two trees with n nodes can be calculated in $\mathcal{O}(n \log n)$ time and in $\mathcal{O}(n)$ space.

Henzinger [39] performed a large scale evaluation on two different algorithms presented by Broder et al. [15] and Charikar [17]. Therefore, a set of 1.6 billion pages have been analyzed by the author. It is stated that both of the approaches do not work very well for detecting similar pages on the same website. Both algorithms strip the HTML tags in order to retrieve only the real text of the website, thus structural elements are ignored. Broder et al.'s approach is based on producing shingles as in Buttler's *Path Shingle* algorithm except for the usage of words instead of HTML elements. Charikar's algorithm produces projections of randomly chosen tokens into multidimensional vectors and utilizes cosine similarity to measure the resemblance of two vectors, i.e. pages. However, Henzinger tried to overcome the drawbacks of both algorithms by combining them and executing them sequentially. The results show that the precision could be raised from 0.50 (Charikar's algorithm) respectively 0.38 (Broder et al.'s algorithm) to 0.79 (combined algorithm).

In [50], Leitao et al. utilizes a Bayesian network to compute the probability of child nodes to be duplicates when thinking of a hierarchically structured HTML document as a tree. It should be noted that the algorithm branches out very quickly and has a worst case complexity of $\mathcal{O}(n * n')$, where n and n' are the number of nodes of the trees to match. The evaluation shows that the algorithm outperforms an existing approach called DogmatiX [80] in terms of precision and recall when tested with three different data sets including both artificial and real world data.

The phishing detection survey [44] of Khonji et al. summarizes state of the art methods for detecting phishing pages. Some of them utilize algorithms which are based on measuring the similarity of a legitimate website and a page in question, thus are applicable for our purposes as well. One of them is called CANTINA [84] and uses Term Frequency-Inverse Document Frequency (TF-IDF) to get the most relevant terms of both documents. They submit a search query to Google's search engine containing the five terms with the highest TF-IDF and check if the results contain the URL in question. This could be used for generating candidates by adding the *inurl:* Google-keyword with *https* as a parameter, which causes the search engine to consider only URLs containing https. However, in order to calculate the TF-IDF value for a term, it is necessary to have a corpus containing a representative set of documents comprising natural language texts for computing the Inverse Document Frequency. In [84], they used the British National Corpus (BNC) for this purpose. Unfortunately, this means that the proper functioning of the algorithm depends on the language of the corpus and thus it is not flexible enough to compare arbitrary HTML pages.

The authors of [66] suggest a method for comparing HTML source code by matching the attributes of the HTML tags. They loop through all tags and compare the attributes of tags of the same type. Unfortunately, it is not explained if the values of all attributes have to be equal or if it suffices if the set of attributes is the same to count as a positive matched tag. They treat two pages as similar if more than 50% of the tags match in both documents. Moreover, they compare the text of the web pages by computing the cosine similarity of the documents and treat them as similar if the resulting value exceeds 50%.

3.3 Fuzzy Hashing

Fuzzy hashing is also referred to as *similarity preserving hashing* (SPH) and as already mentioned before it is designed to create modification resistant fingerprints. Hence, in contrast to cryptographic hash functions, small modifications to the input should affect the resulting fingerprint only to a certain extent. A frequently stated application of such algorithms is file classification in digital forensics.

One of the first approaches of applying fuzzy hashing for forensic purposes has been published by Kornblum in [45]. The described algorithm *ssdeep* has initially been developed by Dr. Andrew Tridgell [75] and is designed to *"identify known files that have had data inserted, modified, or deleted"*[45] and uses *Context Triggered Piecewise Hashing* (CTPH). In this approach a rolling hash, dependent on the currently processed input, as well as a traditional hash is calculated simultaneously. When the rolling hash shows a certain value, called trigger value, the currently processed part of the input will be hashed by using the traditional hashing algorithm and recorded in the result. In this way a small modification of a file only results in a change of a small part of the fingerprint as the file gets hashed piecewise. The similarity value is calculated by computing a weighted edit distance of two given hashes, dividing it by the sum of the hash lengths and, finally, normalizing it in order to make zero a terrible match and 100 a perfect match. The running time of the hash computation is in $\mathcal{O}(n \log n)$ and the comparison of two hashes in $\mathcal{O}(l^2)$, where l is the length of the hash value, thus relatively small and negligible.

However, Roussev states [67] that *ssdeep* quickly loses granularity when processing big files due to the fixed-length hash and proposes another fuzzy hashing approach called *sdhash*. It is characterized by selecting statistically improbable features and filtering out weak features in order to keep the false positive rate low. A feature is a sequence of bits and should constitute a representative and identifying property of a file. Moreover, SHA-1 has been utilized for hashing the selected features and Bloom filters [8] are used for storing the hashes in a memory efficient way. In order to calculate the similarity of two digests, the contents of the corresponding Bloom filters are compared.

In [11] Breitinger and Baier introduce another fuzzy hashing approach called *bbHash*. They state that their algorithm outperforms *sdhash* in terms of hash length as *bbHash* produces fingerprints with only 0.5% of the input size compared to the 2.6%-3.3% of *sdhash*. Furthermore, they explain that *sdhash* only covers about 80% of the input in contrast to their approach covering the whole file. Generally, *bbHash* utilizes random static byte sequences of fixed size, called building blocks, to compute a fingerprint. Moreover, a sliding window of the same size is shifted over the input data byte by byte.

Each time the window moves forward, the hamming distances [37] of the current content of the window to all of the building blocks is calculated. If the smallest computed distance is lower than a certain threshold, the index of the corresponding building block will be appended to the hash value. For the comparison of fingerprints they refer to the approaches used in *ssdeep* and *sdhash*. However, they also state that the major drawback of *bbHash* is the run time performance due to the huge amount of lookup operations necessary when computing a hash. The same authors made an evaluation of some existing fuzzy hashing technologies in [12] and even skipped *bbHash "as its performance is not acceptable"*[12].

Furthermore, they propose a new variation of the *ssdeep* algorithm called *MRSH-v2* [12]. It is based on *Multi-Resolution Similarity Hashing* (MRSH) which is a modification to *ssdeep* proposed by Roussev in [68]. In *MRSH* the rolling hash function is exchanged with the polynomial hash function *djb2* and MD5 is utilized for traditional hashing instead of *FNV*. Moreover, bloom filters are used for storing the final hash value. *MRSH-v2* switches back to the original rolling hash function as the assumption of the author of *MRSH* that *djb2* outperforms rolling hash with respect to performance turned out to be wrong. In order to increase the performance further, in *MRSH-v2* MD5 is exchanged with *FNV-1a* which is the most current version of *FNV* at the time of writing. As the evaluation shows, *MRSH-v2* outperforms all of the before mentioned algorithms in terms of run time and produces hash values of 0.5% of the length of the input data.

Another similarity preserving hash function, *mvHash-B*, introduced in [10] runs through three phases. In the first phase a majority vote for each byte of the input is performed. This method counts the number of binary ones of the currently processed byte and its neighborhood consisting of a constant amount of surrounding bytes. If the bit count is equal or higher than a certain predefined threshold, the resulting bit will be a binary one, zero otherwise. As a second step, *run length encoding* (RLE) is utilized as a compression algorithm. RLE simply counts the number of identical successive bits and creates a sequence of these bit counters as a result. The third and last step deals with the fingerprint generation using Bloom filters. The runtime performance of this algorithm has been compared to the cryptographic hash function SHA-1 and the SPH functions *sdhash* and *ssdeep* by computing the corresponding hashes of a randomly generated 100MiB file. The results show that *mvHash-B* is slower than SHA-1 by a factor of nearly 2.8 but still faster than the other SPH algorithms. Furthermore, they define a new test called *edit operations ratio* (EOR) measuring the robustness of an SPH algorithm by modifying a copy of the input data step by step until the calculated similarity value drops below a certain threshold. The number of tolerated random insertions, deletions and substitutions divided by the original file length gives the EOR. The comparison of the EOR of *mvHash-B* and *sdhash* shows higher values for *sdhash*, thus it is more robust with the trade-off of higher hash value length.

Breitinger et al. proposed another similarity preserving hash function called *saHash* [13]. It utilizes multiple sub-hash functions, which can be exchanged modularly, and operates in linear time. The single hash functions are not very robust singularly as e.g. the first sub-hash function simply yields the length of the input byte sequence. However, in combination they are capable of computing an SPH very efficiently in terms of runtime. They compared the runtime of different hash functions by generating a random 100MiB file and computing the corresponding hash. The evaluation shows that *saHash* works faster than *ssdeep* and *sdhash*, but slower than SHA-1. Moreover, the authors describe that *saHash* is only capable of calculating the similarity of inputs with similar sizes, thus it is not able to detect fragments.

CHAPTER 4

Methodology

The chosen approach aims at the automatic generation of *HTTPS Everywhere* rules in order to cover a larger set of websites that are protected by this extension, thus e.g. protected against *SSL Stripping* and *Cookie Stealing*. To do so, there are three major tasks to manage. At first, websites that support HTTP as well as HTTPS (we call them *candidates* at this stage) have to be found. As it is possible to display completely different content on HTTP and HTTPS websites located on the very same domain, we have to check if these websites are equal by measuring the similarity of the two websites. Candidates that are similar enough to be treated as equal are called *HTTP-HTTPS equivalents* and are considered when generating the rule set files for *HTTPS Everywhere*.

In the following, the approaches to handle each of these major tasks will be described in detail.

4.1 Candidate Generation

In this thesis we use the term *candidate pair* for a pair of URLs, whereas one URL uses the HTTP protocol and the other one uses HTTPS. Such candidates are possibly equal webpages accessible via different protocols, though this is not sure until their similarity is checked.

Since HTTPS web-servers may run on a different subdomain than the corresponding HTTP server it may be rewarding to find existing subdomains.

One of the simplest methods to generate candidates for a given domain is to guess the corresponding HTTPS-URL. In order to make guessing more efficient, one could try frequently used subdomain names of secured websites and, of course, just the same domain but with *https://* instead of *http://* prepended. In order to get an insight into the webmasters' naming strategies for secure domains, we downloaded the current rule set of *HTTPS Everywhere* and implemented a script which analyzes it. The rule set consists (at the time of writing) of over 13,200 XML-files containing about 15,700 active and 1,800 deactivated rules which totals to about 17,500 single rules. We solely considered activated rules as these are currently valid and used by the *HTTPS Everywhere* extension.

During the evaluation of the most popular subdomain names for secured websites we noticed that there are many rules that redirect the user agent to subdomains named *s3* and *a248.e*. When we took a closer look to some single rule-files it turned out that these rules redirect to Amazon's or Akamai's cloud storages[1], thus even to a different second level domain (*s3.amazonaws.com* respectively *a248.e.akamai.net*). Therefore, we modified the script in order to establish new statistics about commonly chosen subdomains and the amount of rules where the second level domain gets rewritten.

Furthermore, some top level domains (TLDs) comprise special second level domains under which third level domains can be registered. An extreme case is the country code TLD (ccTLD) of the United Kingdom. It was not possible to register a second level domain directly under the *.uk* TLD until June 2014 [59]. Only registering third level domains under a limited number of second level domains, like *.ac.uk* or *.co.uk*, has been possible. Similar regulations have been valid for New Zeeland (*.nz*) [56] and other ccTLDs. In order to consider such peculiarities we used the *public suffix list*[2] which is an initiative of the Mozilla Foundation and contains all known suffixes where a domain name can be registered. By utilizing this list it is ascertained that the correct subdomain is extracted from the URL.

In Table 4.1 the ten most frequently occurring subdomain names in the *to*-attributes of all *HTTPS Everywhere* rules are shown in both absolute and relative numbers. In 40% of the cases there is a redirect to the empty subdomain which means that the secure website is hosted on the base domain. Nearly a quarter of all rules show that a redirect to the subdomain *www* is done. The subdomain $1, deployed in more than 10% of the cases, means that the secure site can be reached on the same subdomain of the base domain (e.g. the secure equivalent of *http://blog.example.com/* is hosted on *https://blog.example.com/*). One could assume that the self-evident subdomain names *secure* and *ssl* would be used very frequent for secured websites, but as the evaluation shows, they occur only to a small extent of a bit beyond 2% in total. The rest of the results only occur in under 1% of the cases, thus they are very seldom and negligible.

For retrieving the correct subdomains of the given URLs Mozilla's *public suffix list* has been utilized. In particular, the python package *publicsuffix 1.0.5*[3] has been used for this analysis.

The results of this evaluation directly affects the candidate generation since the *FrequentSubdomainGenerator* (see Section 5.1) creates candidates for each of the eight most frequently used subdomains.

Another possible approach to find existing subdomains is to perform a DNS zone transfer. This mechanism has been introduced to improve the reliability of the naming system. In this way multiple name servers can be set up and kept in sync by exchanging their DNS entries. We can use this for our purpose by requesting the name server (NS)

[1]`https://aws.amazon.com/s3/`, `http://www.akamai.de/`, retrieved: 2014-12-10

[2]`https://publicsuffix.org/`, retrieved: 2014-12-10

[3]`https://pypi.python.org/pypi/publicsuffix/`, retrieved: 2014-12-10

Subdomain	**Absolute**	**Relative (%)**
	6326	40.42
www	3840	24.54
$1	1742	11.13
secure	285	1.82
cdn	45	0.29
ssl	42	0.27
static	41	0.26
shop	36	0.23
my	35	0.22
images	35	0.22
TOTAL	12427	79.4

Table 4.1: The ten most frequent subdomain names of secured websites

entries of a certain domain and sending an Asynchronous Full Transfer Zone (AXFR) request to them. The name servers will answer with all entries of the requested zone and thus we will also receive A-records which may yield valid website domains. However, unauthorized zone transfers are disabled by configuration very often for security reasons.

Furthermore, there are tools for brute-forcing subdomain names like *SubBrute*[4] or *dnsmap*[5]. These programs are shipped with large word lists and try to resolve each of the word list entries as DNS subdomain of an arbitrary base domain. As the enumeration of a large word list lasts relatively long (over 5 minutes on a laptop running *SubBrute* with its standard word list) this method is not suitable for large-scale evaluations.

4.2 Calculating Website Similarity Values

In order to check if a candidate is an HTTP-HTTPS equivalent, i.e. the given HTTP and HTTPS websites show the same content, we have to measure the similarity of the pages. As many websites are generated dynamically, it is not sufficient to simply compare the source codes character by character to determine their equality. Elements like ads, dynamic hidden tokens, date and time or dynamic reordering of products in a webshop may change the website's source code and visual appearance on each and every request. For that reason it is necessary to utilize algorithms that are tolerant against such minor differences, though they should distinguish between different pages on the same website that may have elements like menus and headers in common. Examples for websites showing dynamically changing content are shown in Figures 4.1 and 4.2. The depicted websites have been retrieved in two subsequent requests. As one can see, they partially show the same content like menu and layout whereas some blocks are substituted by others.

[4]`https://github.com/TheRook/subbrute`, retrieved: 2015-02-04
[5]`https://code.google.com/p/dnsmap/`, retrieved: 2015-02-04

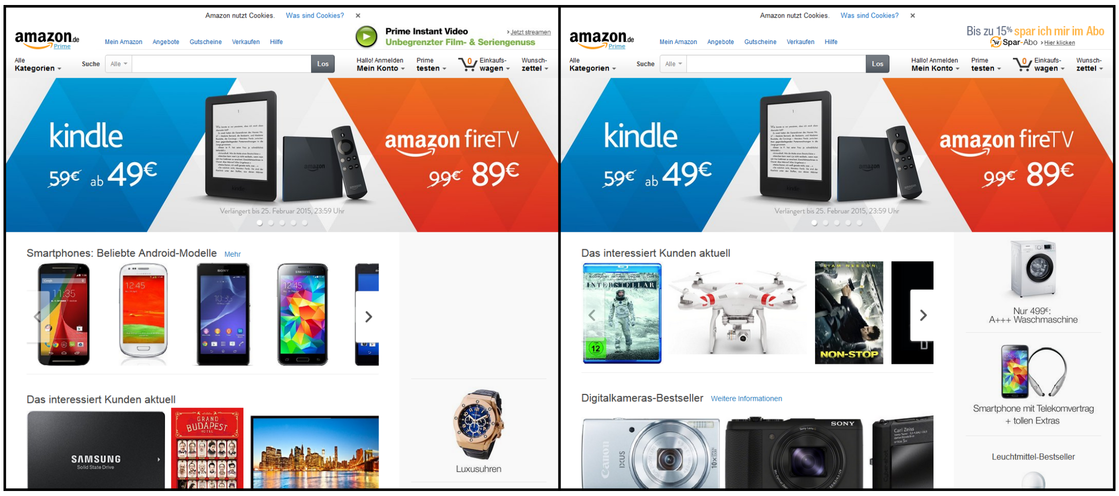

Amazon nutzt Cookies. Was sind Cookies?
Prime Instant Video
Unbegrenzter Film- & Seriengenuss
Bis zu 15% spar ich mir im Abo
kindle
59€ ab 49€
amazon fireTV
99€ 89€
Smartphones: Beliebte Android-Modelle
Das interessiert Kunden aktuell
Digitalkameras-Bestseller
Luxusuhren
Nur 499€: A+++ Waschmaschine
Smartphone mit Telekomvertrag + tollen Extras
Leuchtmittel-Bestseller

Figure 4.1: Results of two subsequent requests to amazon.de

Since our approach needs to overcome the problems arising from this dynamic behavior, we summarized currently used methods for similarity matching in Chapter 3 and discuss their deployment in our approach in the following.

Most of the methods for code similarity matching presented in the survey of Buttler [16] have been tested. The *Tree Edit Distance* (TED) is known to have high requirements in terms of computational power and memory consumption. Nevertheless, we tried to utilize it for website comparison since there is a working Java implementation. *Weighted Tag Similarity* showed the most accurate results in the evaluation of Buttler, though it is mentioned that it may yield not very accurate results when using it for calculating similarity of HTML documents. However, this approach has also been evaluated within this thesis. The proposed path similarity approach [16] has also been adopted and evaluated. *pq-Grams* presented by Augsten et al. [4] approximate the TED in a memory efficient way and requires less computational power. Hence, we tried to utilize pq-Grams for similarity matching. Furthermore, the fuzzy hashing methods *ssdeep* [45] and *sdhash* [67] have been included in our evaluation. Finally, the idea of utilizing *imgSeek* proposed in [38] as well as the application of *LIRe* suggested in [55] have been adopted and evaluated as it can be seen in the next chapters.

In addition, we developed a novel approach for website comparison by using the well-known Levenshtein distance[51]. This metric represents the minimum number of insertion-, deletion- and substitution operations that are required to transform one string into another. Unfortunately, it is computationally expensive to calculate the Levenshtein distance. Its runtime complexity is in $\mathcal{O}(n * m)$, where n and m are the lengths of the strings to compare. However, we utilized this algorithm in our software and tested it in various configurations and with differently preprocessed inputs. The detailed descriptions of these variations can be seen in Section 5.2. Furthermore, the simple image matching approach described by Dr. Neal Krawetz in [47] has been implemented and tested for our purpose. The first step of this algorithm requires shrinking the screenshot of the website

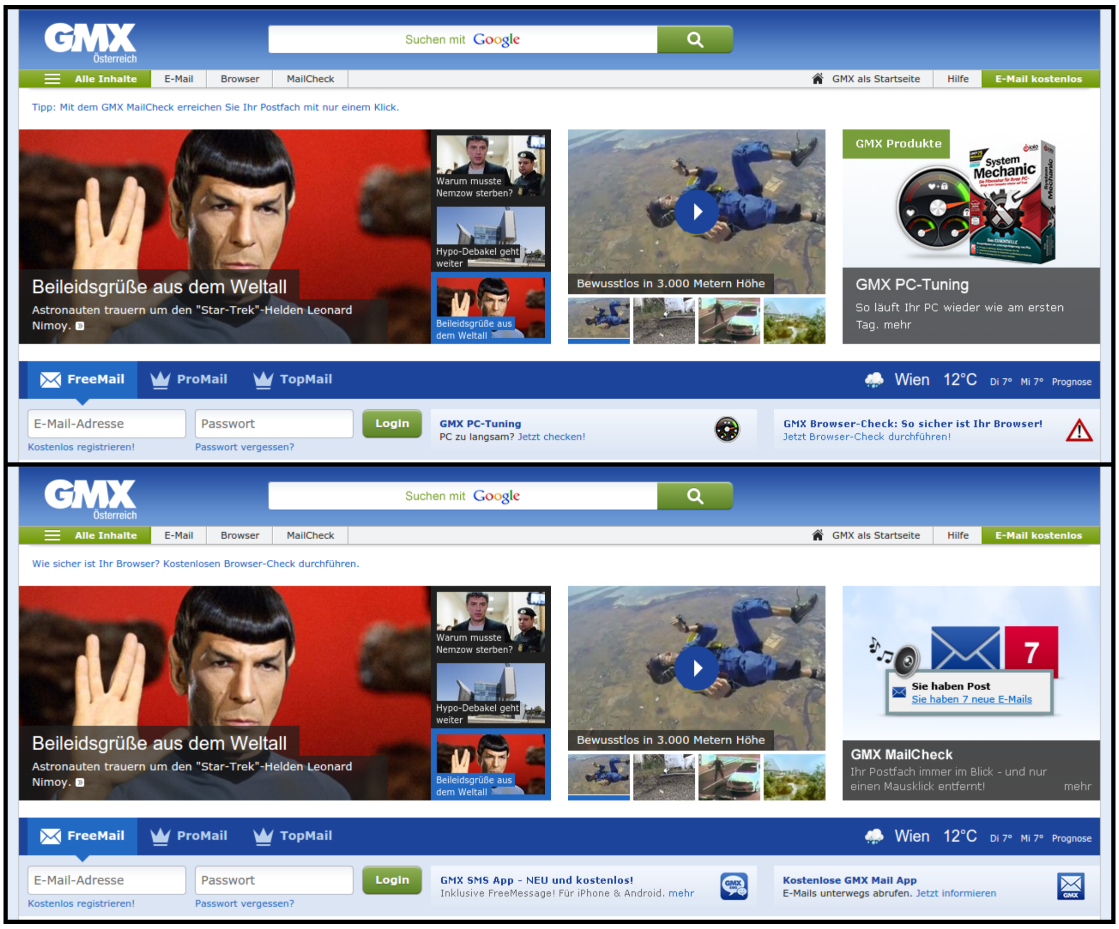

GMX Österreich
Suchen mit Google
Alle Inhalte
E-Mail
Browser
MailCheck
GMX als Startseite
Hilfe
E-Mail kostenlos
Tipp: Mit dem GMX MailCheck erreichen Sie Ihr Postfach mit nur einem Klick.
Beileidsgrüße aus dem Weltall
Astronauten trauern um den "Star-Trek"-Helden Leonard Nimoy.
Warum musste Nemzow sterben?
Hypo-Debakel geht weiter
Beileidsgrüße aus dem Weltall
Bewusstlos in 3.000 Metern Höhe
GMX Produkte
GMX PC-Tuning
So läuft Ihr PC wieder wie am ersten Tag. mehr
FreeMail
ProMail
TopMail
Wien 12°C Di 7° Mi 7° Prognose
E-Mail-Adresse
Passwort
Login
Kostenlos registrieren!
Passwort vergessen?
GMX PC-Tuning
PC zu langsam? Jetzt checken!
GMX Browser-Check: So sicher ist Ihr Browser!
Jetzt Browser-Check durchführen!
GMX Österreich
Suchen mit Google
Alle Inhalte
E-Mail
Browser
MailCheck
GMX als Startseite
Hilfe
E-Mail kostenlos
Wie sicher ist Ihr Browser? Kostenlosen Browser-Check durchführen.
Beileidsgrüße aus dem Weltall
Astronauten trauern um den "Star-Trek"-Helden Leonard Nimoy.
Warum musste Nemzow sterben?
Hypo-Debakel geht weiter
Beileidsgrüße aus dem Weltall
Bewusstlos in 3.000 Metern Höhe
Sie haben Post
Sie haben 7 neue E-Mails
GMX MailCheck
Ihr Postfach immer im Blick - und nur einen Mausklick entfernt! mehr
FreeMail
ProMail
TopMail
Wien 12°C Di 7° Mi 7° Prognose
E-Mail-Adresse
Passwort
Login
Kostenlos registrieren!
Passwort vergessen?
GMX SMS App - NEU und kostenlos!
Inklusive FreeMessage! Für iPhone & Android. mehr
Kostenlose GMX Mail App
E-Mails unterwegs abrufen. Jetzt informieren

Figure 4.2: Results of two subsequent requests to gmx.at

to a size of 8x8 pixels and converting it to greyscale. Then the mean greyscale value of the shrinked image is calculated and used as threshold. Finally, the 64-bit fingerprint is calculated by traversing all pixels and appending a binary 1 to the result if the greyscale value of the currently visited pixel exceeds the threshold, 0 otherwise. Two fingerprints are compared by calculating their hamming distance [37], i.e. counting the amount of differing bits of the two hashes.

In the approach proposed in [73] websites with similar layouts are treated as positive match. Our use case requires distinguishing webpages that may have some elements in common. Therefore, this approach is not suitable for our purpose. The visual-similarity approaches presented in [35], [57], [18] and [83] look promising, though there is no source code publicly available that could be used within our solution. Since implementing them would mean too much effort we did not evaluate these approaches. The Fast Fourier Transformation (FFT) approach is, according to Buttler, less accurate than other tested algorithms in [16] and therefore has not been tested. *CANTINA* [84] utilizes *Term Frequency-Inverse Document Frequency* (TF-IDF) which requires a corpus of documents to work. Since we want to classify websites disregarding their language, implementing this approach would probably be unrewarding. The Bayesian approach of Leitao et al. [50] branches out very quickly on larger documents, thus having higher requirements on processing power and memory. Additionally, there is no working implementation available. As the effort would probably exceed the benefit, we did not utilize Leitao et al.'s procedure. The same applies to the algorithms of Broder et al. [15] and Charikar [17] since we could not find any publicly available implementations. The approach described in [66] is explained not very accurate, thus reprogramming would possibly yield different results than stated in the paper and was not done therefore. The fuzzy hashing approach *bbHash* [11] has been skipped due to known performance issues. Even its authors did not evaluate it in a second paper [12] because of its infeasible runtime. The algorithms presented in [10] and [13] could not be tested, as we were unable to compile the source code which was initially written on a Microsoft Windows system by using Microsoft Visual Studio[6]. Moreover, there is no documentation on how to compile and run the software on a UNIX based operating system; hence we could not resolve missing dependencies.

Details regarding the implementation of the evaluated approaches can be seen in Chapter 5.

4.3 HTTPS Everywhere Rule Generation

After the candidates are generated and their similarity values are calculated, rule files for *HTTPS Everywhere* can be created. Since there are only the similarity values of all checked candidates available, the decision if two websites are similar enough to be considered as HTTP-HTTPS equivalent has to be made at this stage. Therefore, all of the different matchers have to be evaluated in order to choose the most suitable matchers and determine their thresholds. The procedure of this evaluation can be seen in Chapter 6.

Each candidate pair that exceeds the thresholds of the chosen matchers will be incorporated into the rule set. Furthermore, some websites perform automatic redirects from HTTP to HTTPS. These redirects will also be considered and appropriate rules are included in the result since it still makes sense that *HTTPS Everywhere* initiates the redirect instead of the web-server (see Section 2.2).

Obviously, redirects from HTTPS to HTTP (called *downgrade redirects*) have to be ignored and are not included in the rule set.

In the following the structure of *HTTPS Everywhere* rule files is explained in detail. Listing 4.1 shows an example for a currently valid rule set-file for the website of the Vienna University of Technology. The semantics of the different XML tags included in such files is described at [29] and will be summarized in the following. The root tag of each rule set file is `ruleset` that contains a `name`-attribute and optionally a `default_off`-attribute denoting that the contained rules are disabled.

Moreover, a rule set file comprises one or more `target`-tags each containing one `host`-attribute. These define the domains affected by the rules contained in this rule set.

[6]`http://www.visualstudio.com/`, retrieved: 2015-02-19

As it can be seen in the provided example, it is possible to use wildcards (*) within the `host`-attribute, though only one per attribute is allowed. If the wildcard is on the left of the domain like in the example (**.tuwien.ac.at*), it will match arbitrary long subdomains like *www.zid.tuwien.ac.at*.

The optional `exclusion`-tag has one `pattern`-attribute that contains a JavaScript regular expression describing URLs that should be ignored despite the fact that they may be included in rewriting rules. Such exclusions are helpful when parts of websites are available via HTTP but not via HTTPS.

Moreover, it is possible to explicitly set the secure flag (see Section 2.2.1) for cookies provided by a website. This can be done by specifying a `securecookie`-tag containing a `host`- and a `name`-attribute. The former expresses the hosts that should be affected by the tag, whereas the latter shows the name of the cookies that should be secured. Both attributes have to be expressed as regular expressions.

The core functionality of *HTTPS Everywhere*, namely URL rewriting, is specified by using the `rule`-tags. They comprise a `from`- as well as a `to`-attribute. When the user navigates to an URL that is described by a regular expression in a `from`-attribute, the extension will rewrite the URL according to the corresponding `to`-attribute. The second rule in the given example shows the sequence *$1* inside of the `to`-attribute. This way it is possible to make URL rewriting more generic as these characters get replaced by the first bracketed part of the regular expression. In our example the second rule rewrites all of the given OR-ed subdomains to the same subdomain as *$1* gets replaced by the appropriate subdomain of the original URL. For example: *http://ui.zid.tuwien.ac.at/* would get rewritten to *https://ui.zid.tuwien.ac.at/*. For a detailed explanation of JavaScript regular expressions see [21] or [79].

```
<ruleset name="Vienna University of Technology (partial)">
  <target host="tuwien.ac.at"/>
  <target host="*.tuwien.ac.at"/>
    <exclusion pattern="^http://www.zid\.tuwien\.ac
      \.at/sts/(?!campussoftware|studentensoftware)"/>

  <securecookie host="^(?:\.mail\.student|\.webmail|www)
  \.tuwien\.ac\.at$" name=".+"/>

  <rule from="^http://(?:www\.)?tuwien\.ac\.at/"
    to="https://www.tuwien.ac.at/"/>
  <rule from="^http://(mail|pop|mail\.student|webmail|
    (?:ui|www)\.zid|(?:mail|webstats)\.zserv)\.tuwien\.ac\.at/"
    to="https://$1.tuwien.ac.at/"/>
</ruleset>
```

Listing 4.1: Example HTTPS Everywhere rule set-File (comments omitted)

Moreover, rules can be used to downgrade a connection from HTTPS to HTTP. This is sometimes necessary as some websites show broken links when accessed with using *HTTPS Everywhere* as the referenced resources are only available via HTTP. In order to avoid unintentional downgrading from HTTPS to HTTP, the extension does not allow it without specifying an additional `downgrade`-attribute within the corresponding `rule`-tag.

CHAPTER 5

Software Design

Basically, our software comprises four highly decoupled components, namely

- Candidate Generator,
- Candidate Checker,
- Aggregator and
- Rule Generator.

In order to make the software highly scalable, we decided to interconnect the first three components by using message queueing. This approach allows starting up multiple *Candidate Generators* acting as data sources and also various *Candidate Checkers* acting as data sinks. These workers read candidates from the *candidates* queue, process them and write the results to the *results* queue. The candidate pairs are published to the queue within work packages, such that particular candidates can be forced to be processed by one and the same worker. The *results* queue is read by the *Aggregator* that stores the calculated similarity values of the candidates into the database. Finally, the *Rule Generator* creates the rule set for the *HTTPS Everywhere* extension.

All components can be distributed among different hosts, such that the work load is balanced between them and performance gets increased. Furthermore, such a loosely coupled design provides the possibility to exchange single components of the program, so for example a new *Candidate Generator* can be implemented in Python and connected to the rest of the software by simply publishing messages to the corresponding message queue. The coarse software architecture is depicted in Figure 5.1.

We decided to use Java[1] 7 for the implementation of the single components and a RabbitMQ[2] 3.2.4 server for exchanging messages. To store the results we utilized the

[1]`https://www.java.com/`, retrieved: 2015-02-18
[2]`https://www.rabbitmq.com/`, retrieved: 2015-02-18

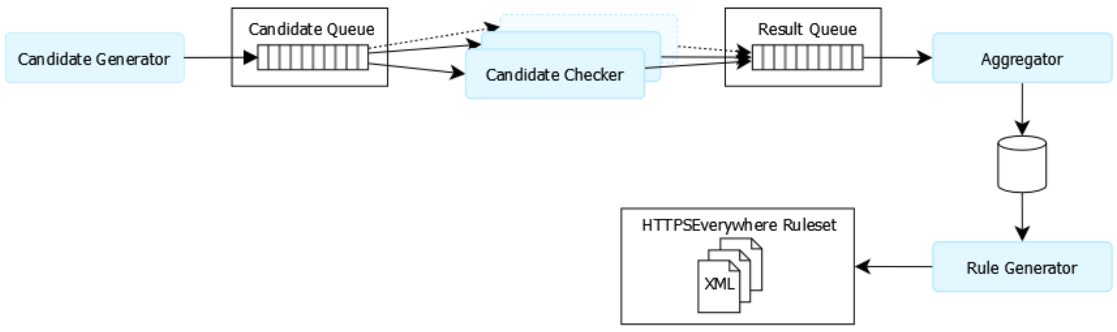


Figure 5.1: Software Architecture

relational database PostgreSQL[3] 9.3.6. The rest of the utilized libraries are mentioned in the sections of the corresponding component that uses them.

As possibly already assumed, the software is not intended to run on the clients using *HTTPS Everywhere*, but on a server infrastructure that generates new rules for the users of the extension.

5.1 Candidate Generator

The *Candidate Generator* is responsible for creating potential candidates of HTTP-HTTPS equivalents. Simply put, this component generates pairs like (*http://example.com/*, *https://secure.example.com/*) which could possibly show equal content. However, it does not check if the generated candidates really have equal content. For the rest of the document candidate pairs are denoted in the same way like it has been done just before: (*http-url*, *https-url*).

As already mentioned, the candidate pairs are bundled to work packages before publishing them to the queue. This can be beneficial if it is sufficient to find only one HTTPS equivalent for one corresponding HTTP-URL. Suppose that the following two candidates have been generated: (*http://example.com/*, *https://example.com/*) and (*http://example.com/*, *https://secure.example.com/*). By sending them bundled, such that only one worker is charged with the task of processing both related candidates, it is possible to stop working on this work package when a positive match has been found. Since it is only possible to have one HTTPS target for every HTTP-URL within the *HTTPS Everywhere* rule set, it suffices to have one positive match. Furthermore, skipping the rest of the candidates means saving time, i.e. if (*http://example.com/*, *https://example.com/*) are equivalent, there will be no need for inspecting all future candidate pairs that show *http://example.com/* as an HTTP-URL as well. If these would not get bundled, it would be very likely that different *Candidate Checkers* receive candidates containing one and the same HTTP-URL and process them without knowledge about previous similarity calculations.

[3]`http://www.postgresql.org/`, retrieved: 2015-02-18

In addition to the possibility of skipping candidates if an HTTP-HTTPS equivalent has been found, the packaging of candidate pairs yields the additional advantage that the same HTTP-URL potentially occurs multiple times inside of one work package. This makes caching very effective.

The different chosen approaches for generating candidates are organized within *Generators* in our software. The common abstract super-class of the concrete implementations guarantees maintainability and extensibility. All *Generators* expect an HTTP-URL as input and return a list of candidate pairs. The following *Generators* have been implemented:

TestSetGenerator The *TestSetGenerator* simply returns the candidates that have been chosen as a training set for determining the accuracy, performance and threshold of the implemented matchers (see Chapter 6). Hence, this Generator is not designed for the large-scale evaluation.

FrequentSubdomainsGenerator This Generator bases on the evaluation of frequently used subdomains (see Section 4.1 and Chapter 6) for HTTPS secured websites. The host-part of the URL is extracted using Java's *URL* class and is passed to the Java implementation of Mozilla's *public suffix list*[4] in order to get the registered domain. Then candidates comprising the eight most frequently used subdomains are generated. The generated candidates for the example input *http://example.co.uk/* can be seen in Table 5.1. Since there is no subdomain in this case, only seven candidate pairs are generated.

HTTP-URL	**HTTPS-URL**
http://example.co.uk/	https://example.co.uk/
http://example.co.uk/	https://www.example.co.uk/
http://example.co.uk/	https://secure.example.co.uk/
http://example.co.uk/	https://cdn.example.co.uk/
http://example.co.uk/	https://ssl.example.co.uk/
http://example.co.uk/	https://static.example.co.uk/
http://example.co.uk/	https://shop.example.co.uk/

Table 5.1: Candidates generated for *http://example.co.uk/*

SimpleZoneTransferGenerator The *SimpleZoneTransferGenerator* tries to perform a DNS zone transfer for the supplied host and generates candidate pairs comprising the initially passed HTTP-URL and HTTPS-URLs assembled from the retrieved DNS A-records. If unauthorized zone transfers are not enabled on the respective DNS server, no candidates will be generated. For the implementation of this method, *dnsjava 2.1.6*[5] has been used to perform the zone transfer.

[4] `https://github.com/whois-server-list/public-suffix-list`, retrieved: 2015-03-06
[5] `http://www.dnsjava.org/`, retrieved: 2015-02-24

NestedZoneTransferGenerator This Generator is able to invoke other Generators when a DNS zone transfer succeeded in order to generate even more candidates. It supplies the retrieved A-records to one or more other Generators, accumulates the results and returns all generated candidate pairs.

After the candidates for a set of URLs have been generated they are combined to work packages by grouping them with respect to their HTTP-URLs. Finally, the *Candidate Generator* publishes the work packages to the *candidates* message queue.

5.2 Candidate Checker

The task of calculating similarity values of candidate pairs is taken over by the *Candidate Checker* that utilizes visual and/or code similarity techniques. This presumes that some kind of automated browser is utilized for gathering HTML code and screenshots of the webpages in question. In our implementation we made use of Selenium[6] with Ghost Driver[7] as a WebDriver implementation. Ghost Driver utilizes PhantomJS[8] as a backend which is a headless browser relying on WebKit[9].

The software has been designed in a way that the PhantomJS browser can easily be exchanged by other alternatives. Therefore, a Java interface is utilized that can be used for integrating other browsers. Moreover, it is not even absolutely necessary to utilize an automated browser since the abstraction allows deploying arbitrary procedures for fetching the source code and screenshot of a website.

The *Candidate Checker* reads work packages from the *candidates* queue, processes them and publishes the calculation result to the *results* queue. In addition to deploying these workers onto multiple servers, it is possible to start an arbitrary amount of threads. Each of them processes its own work packages by using its own browser instance. The limitation for the maximum number of threads accrues from the available computational power, memory and network bandwidth.

The different approaches that have been selected for calculating the similarities of websites (see Chapter 4) are organized in so called *matchers* in our implementation, having one common interface to make the design modular and extensible. Matchers that should be used for similarity calculation can be easily added to and removed from our software. In the following the implementations of the various tested approaches are described in detail:

FuzzyImageMatcher The FuzzyImageMatcher bases on the approach described in [47]. For downscaling of the screenshot made by PhantomJS, we utilized the Java standard scaling method *Image.getScaledInstance()* with the *SCALE_FAST* parameter. The actual similarity value is calculated by dividing the hamming distance[37] of the websites' fingerprints by the maximum possible hamming distance of 64 (as each

[6]`http://www.seleniumhq.org/`, retrieved: 2015-02-24
[7]`https://github.com/detro/ghostdriver`, retrieved: 2015-02-24
[8]`http://phantomjs.org/`, retrieved: 2015-02-24
[9]`https://www.webkit.org/`, retrieved: 2015-02-24

fingerprint is 64 bit long) and subtracting it from 1 ($sim = 1 - \frac{HammingDistance(FP1,FP2)}{64}$).

LevenshteinMatcher The first attempt we made to utilize the Levenshtein distance within a matcher, was a very basic one. It simply took the whole HTML documents and calculated their distance. The implementation of the Levenshtein distance algorithm has been taken from [82]. Since the algorithm is not very efficient in terms of runtime, the calculation lasts relatively long even for websites with small source code sizes. Since the similarity calculations for only 1,000 websites with an average source code size of 60 KiB would last about 10 hours, we decided not to use it within the final software. A brief runtime evaluation is shown in Chapter 6.

LevenshteinCodeMatcher Since the runtime performance of the Levenshtein-Matcher is poor due to the length of the input string, we reduced the number of characters by solely considering the structural elements of the HTML documents. As textual content is stripped out, the size of an HTML file decreases and makes the distance calculation faster. One problem with this approach is that the length of the name of a tag and its attributes influence the Levenshtein distance if this tag is missing or permuted in the second document, as each and every letter of the HTML tag is incorporated. The solution for this problem additionally improves the performance of this matcher. Each tag gets an identifier in form of a consecutively numbered integer assigned, disregarding the attributes contained by the respective tag. This means that e.g. the tags `<p>` and `<p align="center">` are tagged with the same number. After all tags are numbered, a string representing the HTML document is constructed by recording each occurrence of a tag with its corresponding identifier stored in one character of the string. This has two positive effects. Firstly, each Levenshtein-operation on one tag is counted only once disregarding the tag's string length. Secondly, the length of the input is shortened further which directly affects the processing time. It should be noted that neither the attributes of the tags, nor the corresponding end-tags of the elements are considered in this approach. For the implementation jsoup[10] has been used for extracting the HTML tags out of the full HTML document.

LevenshteinCodeMatcherWithEndTags As already mentioned, the Levenshtein-CodeMatcher does not consider HTML end-tags in the distance calculation. In order to measure the impact of end-tags on similarity matching, an additional matcher has been created that encodes start-tags as well as end-tags into the string containing the packed HTML structure. In Chapter 6, the two matchers are compared and the impact of considering end-tags is evaluated.

LevenshteinTextMatcher The counterpart of the LevenshteinCodeMatcher(WithEndTags) is represented by the LevenshteinTextMatcher that, unlike to the former, solely takes the text contained in the HTML document into account. Thus, HTML tags are stripped out and the Levenshtein distance is computed only for the textual elements. Since

[10] `http://jsoup.org/`, retrieved: 2015-02-18

websites sometimes comprise much text, the calculation may last relatively long in comparison with other tested algorithms. The average calculation durations needed for the candidates in the training set has been evaluated and can be seen in a later chapter.

LevenshteinWordMatcher In order to speed up the LevenshteinTextMatcher we developed a faster approach that operates on the same data, i.e. the textual content of a webpage. In principle, this improvement is similar to that deployed at the LevenshteinCodeMatcher. Each word in the text gets a numeric identifier assigned that is used for assembling a new string representing the corresponding HTML document. Since there is an infinite amount of possible words, it may occur that the same identifiers are assigned to multiple different words. This possibly seems problematic but since exact matching is no requirement, it is legitimate in this case. As it can be seen in Chapter 6, the developed procedure achieves its goal and gives this matcher the desired speed-up.

PathSimilarityMatcher The PathSimilarityMatcher bases on the approach presented in [16] and traverses the HTML tree by using jsoup in order to construct the path-strings (see Section 3.3 resp. Listing 3.1). In order to keep the retrieval of the path strings out of the set of paths fast, we utilized a HashMap.

PQGramMatcher The approach of using *pq-Grams* [4] for measuring similarity of HTML trees has been implemented by using jqgram[11]. It is a nodejs[12] implementation that requires the HTML trees to be in JSON notation. Therefore, an additional file is assembled dynamically and is taken as input for nodejs. Again jsoup is utilized for the construction of the HTML tree.

SSDeepMatcher The SSDeepMatcher utilizes the *ssdeep* [45] algorithm to create fuzzy hashes of the websites to compare. In particular, a Java port of *ssdeep*[13] has been deployed to match the whole HTML documents. It computes the fingerprints of both documents and uses the already described (see Chapter 3) fingerprint comparison method to compute the similarity of the webpages.

SSDeepCodeMatcher This matcher applies the *ssdeep* algorithm to the structural elements of the webpage's code. In order to solely consider the tags of the HTML documents and to keep the length of the input for *ssdeep* short, the tags get encoded as integers as it is done at the LevenshteinWordMatcher and the LevenshteinCodeMatcher. As at the latter, HTML end-tags are not considered, thus are not included in the constructed string.

SSDeepImageMatcher The SSDeepImageMatcher utilizes *ssdeep* to compare the screenshot images of the possibly equivalent websites. It uses the standard Java imple-

[11] `https://www.npmjs.com/package/jqgram`, retrieved: 2015-02-19
[12] `http://nodejs.org/`, retrieved: 2015-02-19
[13] `https://github.com/openplanets/bitwiser/blob/master/bitwiser-core/src/main/java/eu/scape_project/bitwiser/utils/SSDeep.java`, retrieved: 2015-02-19

mentation BufferedImage to get the bytes of the image in ARGB format and uses this array as input for the *ssdeep* Java port.

SSDeepTextMatcher The counterpart of the SSDeepCodeMatcher is represented by the SSDeepTextMatcher which solely considers the text of HTML documents. Like the LevenshteinTextMatcher and the LevenshteinWordMatcher, it utilizes the jsoup library for stripping out the HTML code.

SDHashMatcher The SDHashMatcher takes the whole HTML document as input for the *sdhash* [67] algorithm. Since there is no Java port of *sdhash*, we execute the command line tool provided by the algorithm's inventors and parse its output to retrieve the similarity values.

TEDMatcher This matcher computes the Tree Edit Distance (TED) of the HTML trees of the two documents and calculates their similarity by setting the TED value in relation to the number of HTML elements (= tree nodes) comprised by the document with the bigger HTML tree. The algorithm used for calculating the TED is the already mentioned RTED algorithm (see Chapter 3). The Java implementation has been taken from the website of the algorithm's inventors[14]. In order to compose the tree needed as input for the RTED algorithm, we used the jsoup library once more.

WeightedTagSimilarityMatcher In order to compute the *Weighted Tag Similarity* presented by Buttler [16], the number of occurrences of all HTML tags has to be counted. Again, the *jsoup* library has been utilized for identifying and counting the different elements.

After the similarity calculation of an arbitrary selection of one or more different matchers the results are published to the *results* queue.

5.3 Aggregator

The *Aggregator* is used for persisting the results of the similarity calculations made by the *Candidate Checker* into a database. We chose to utilize Hibernate[15] as an object-relational mapper and, as already mentioned before, PostgreSQL[16] as the corresponding relational database.

The *Aggregator* has been introduced in order to keep persisting of the gathered data simple. The possibly distributed *Candidate Checkers* will not have to be redeployed if the database backend or the database scheme changes since the *Aggregator* is a central component usually deployed only once.

[14]http://www.inf.unibz.it/dis/projects/tree-edit-distance/index.php, retrieved: 2015-02-24
[15]http://hibernate.org/, retrieved: 2015-02-24
[16]http://www.postgresql.org/, retrieved: 2015-02-24

5.4 Rule Generator

The *Rule Generator* is responsible for creating the rule set for *HTTPS Everywhere* out of the gathered data. Therefore, appropriate data sets are fetched from the database again by using Hibernate. In order not to fetch all available datasets, the thresholds of the chosen algorithms are already considered in the database query. Each result set comprises the following fields:

- **originalUrl1:** HTTP-URL that has been initially requested
- **redirectedUrl1:** URL that the browser has been redirected to when it tried to access *originalUrl1* or *NULL* if there was no redirect
- **originalUrl2:** HTTPS-URL that has been initially requested
- **redirectedUrl2:** URL that the browser has been redirected to when it tried to access *originalUrl2* or *NULL* if there was no redirect
- **matcher:** Name of the matcher that has been used for similarity calculation
- **similarity:** Computed similarity value

The steps that are taken for generating the rules out of the result sets stored in the database can be seen in Algorithm 5.1. The output of this algorithm is supplied to another helper-class that groups all generated rules by the base domain of the URL stored in the `from`-attribute. Again, the *public suffix list* has been utilized to determine the correct domain. After the grouping operation, rules belonging to the same website are bundled and can now be stored within one XML rule-file. This storing-task is taken over by another class (*RuleSetWriter*) that utilizes *StAX Utilities*[17] for writing the rules to the disk. Moreover, the *RuleSetWriter* escapes the `from`- and `to`-attributes accordingly, such that they meet the requirements of both the JavaScript RegExp syntax and, certainly, the *HTTPS Everywhere* extension.

5.5 Performance Improvements

As we are gathering data on a large scale, we tried to make some improvements regarding performance in order to speed up the crawling process.

A cache has been introduced at the *Candidate Generator* and *Candidate Checker* in order not to face the same redirects again and again. As the *Candidate Generator* usually produces multiple candidates containing one and the same HTTP-URL, it is possible that all of them cause the very same redirect when accessed by the *Candidate Checker*. For example suppose that the *Candidate Generator* creates two candidate pairs (*http://example.com/*, *https://example.com/*) and (*http://example.com/*, *https://www.example.com/*). When the *Candidate Checker* accesses *http://example.com/*

[17]`https://java.net/projects/stax-utils/pages/Home`, retrieved: 2015-03-10

Algorithm 5.1: Rule generation

```
Input: ResultSet[ ] resultsets
Result: Rule[ ] ruleset
forall the rs in resultsets do
    Rule r;
    // redirect from HTTP to HTTPS occurred?
    if rs.redirectedUrl1 starts with 'https://' then
        r.from := rs.originalUrl1;
        r.to := rs.redirectedUrl1;
        ruleset.append(r);
        continue
    end
    // downgrade redirect from HTTPS to HTTP occurred?
    if rs.redirectedUrl2 starts with 'http://' then
        continue
    else
        r.from := rs.originalUrl1;
        r.to := rs.originalUrl2;
        ruleset.append(r);
        if rs.redirectedUrl1 is not NULL then
            r.from := rs.redirectedUrl1;
            r.to := rs.originalUrl2;
            ruleset.append(r);
        end
    end
end
```

the first time a redirect to *http://example.com/start/* occurs. As this will change very unlikely during one run, it is stored into the cache in order to access the page the *Candidate Checker* will be redirected to directly. Furthermore, domains which refuse connections to the corresponding ports of HTTP or HTTPS protocol, i.e. unreachable websites, get also cached in order not to probe the same hosts on open ports again and again. The same is true for websites that keep the browser in loading state for over one minute, thus causing a load timeout.

Moreover, an additional check for the currently utilized matcher types has been introduced in order to fetch the source code solely if there is a matcher that needs the source code to calculate a similarity value. The same is true for matchers requiring screenshots.

The introduction of this cache for redirects and unreachable sites resulted in a considerably large speed-up. With using a test set containing the top 20 of Alexa's top sites[18] and using the *FrequentSubdomainGenerator* the first run without utilizing a cache took over 20 minutes. The second run has been done on the same data set with the cache enabled and it lasted for not even 9 minutes which means an improvement of nearly 57%. PhantomJS defaults to a configuration that turns the browser cache off. Enabling this caching-option leaded to an execution time of about 3.5 minutes which means an additional speed-up of nearly 60%. These measurements have been performed on a notebook with 4GiB memory and an Intel i5 processor.

The *Candidate Generator* fetches the URLs of a candidate sequentially. Based on the preceding example it will load *http://example.com/* first and *https://example.com/* subsequently. If the *FrequentSubdomainGenerator* has been utilized for candidate generation, it is very likely that the next candidate pair also shows *http://example.com/* as one of its URLs. Since the browser loaded a different website in between these two equal requests (namely *https://example.com/*) it has to load it from its browser cache and execute all embedded scripts again which produces computational overhead. Hence, a source code cache has been introduced which stores only a few of the last responses to avoid this effect.

Another attempt to improve the performance of the *Candidate Checker* further was to utilize a hybrid crawler which makes use of both Java's HttpURLConnection and the PhantomJS browser. The former is solely used for fetching the source code of a website, the latter for taking screenshots of a website. This increases the performance of flexible approaches that do not need screenshots on every website access since the HttpURLConnection solely requests the HTML code without embedded objects like JavaScript code and images. Moreover, no website rendering engine gets started, thus computational and memory requirements decrease. Certainly, this only applies if the utilized matchers require no screenshot of the crawled webpages, otherwise a rendering engine is necessary which means a real browser like PhantomJS has to be deployed.

One problem with using only HttpURLConnection for fetching the HTML code is that this class does not automatically follow redirects from HTTP to HTTPS websites. However, this can be implemented manually by retrieving the HTTP status code of the response and considering the *Location*-header field as the new destination URL. Another, more problematic, issue is that redirects can be also done by using an HTML `meta`-tag or operations on JavaScript's *window.location* property. Unfortunately, there is no realistic possibility to evaluate such expressions without utilizing a HTML and JavaScript engine. For that reason the hybrid crawler could not be used as a reliable source for HTML code.

Another improvement has been implemented on the overall process by introducing an additional validation inside of the *Candidate Generator*. It checks the candidates on reachability before writing the jobs into the message queue. Hence, it tries to open a connection to TCP port 80 respectively 443 and immediately closes it after it has been established. This reduces the work load for the *Candidate Checkers* as they do not have to try to connect to unreachable resources which means an additional speed-up of the overall process. The reachability-check has been implemented in the common super-class

[18] `http://www.alexa.com/`, retrieved: 2015-03-12

of the Generators in order to make this additional functionality available for all currently available Generators and for future extensions.

CHAPTER 6

Evaluation

6.1 Candidate Generation

As described before, the *Candidate Generator* performs a connect-check of all generated URLs before it hands them over to the *Candidate Checker* via the message queue. When testing many domains the timeout used when connecting to a certain domain has a large impact on the overall runtime. On the one hand, a long timeout value prolongs the candidate generation to a large extent. On the other hand, short timeouts could lead to false negatives, i.e. the server could have been reached but the response took longer than the configured timeout allowed.

In order to find a suitable connect timeout, we took a random set of 1,000 pages out of the Alexa's top million and varied the connect timeout when generating candidates. In particular, we utilized the *FrequentSubdomainGenerator* and tried timeouts between one and ten seconds. Interestingly, the first iteration with a timeout of one second took longer than the second run with a timeout of two seconds. We found that this paradoxical result originates from DNS resolution and caching. When a system requests a DNS server to resolve an address, the client probably caches the result to serve future requests faster. In order to get rid of this DNS resolving time we made an additional disregarded run on the same set of domains before running the whole test with different timeouts. The evaluation of the candidate generation with using different timeouts showed that the overall time needed to generate candidates linearly increases with the chosen timeout. This behavior is depicted in Figure 6.1. Moreover, we considered the number of candidates found when running the tests in order to detect the optimal timeout value that cause a minimum of missed candidates but still acceptable timeouts. The results depicted in Figure 6.2 show that the timeout of one second might be too short for some servers to answer since other tried timeouts yield more candidates. However, higher timeouts do not necessarily mean higher precision as it can be seen when taking a look at the number of candidates found when waiting eight or nine seconds for a response. These low values possibly arise from variations of the internet connection. Therefore, we chose to set the timeout to three

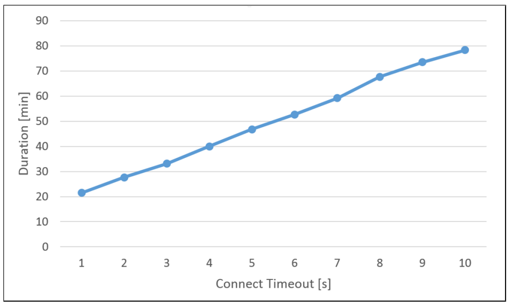


Figure 6.1: Overall Duration

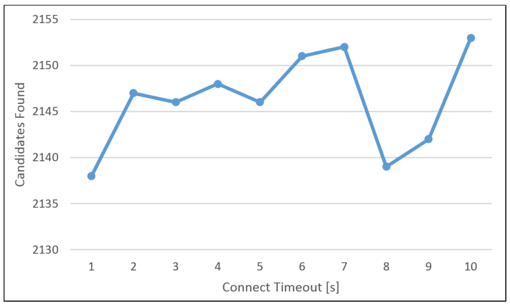


Figure 6.2: Candidates found

seconds for our large-scale evaluation since this leads to an acceptable runtime and still accurate results.

6.2 Candidate Checking

In order to find the most accurate and performant matchers we defined a training set of websites containing *negative*, *similar but negative* and *positive* candidate pairs and made 50 test runs with all implemented matchers. We averaged the results of the similarity calculations for each candidate pair to be able to compare the matchers' classification accuracy and performance. The multiple test runs on the same set of candidates have been performed in order to get different versions of dynamically generated websites and thus more realistic conditions. The websites contained in the training set can be seen in Appendix A. Candidates in the category *negative* show completely different content in terms of layout and textual content. *Similar but negative* candidates share either textual or layout characteristics. *Positive* candidates contain websites that should be treated as HTTP-HTTPS equivalents but may also comprise dynamically generated elements.

Some of the visual similarity algorithms would have been hard to integrate directly into our software because of missing Java bindings. For that reason we used PhantomJS (outside of our software) to store screenshots of all websites in our training set and tested the algorithms manually. In particular, this has been done for *imgSeek* and *LIRe*.

Since *imgSeek* is a stand-alone program, the duration of the calculations could not be measured as there is only a GUI available for retrieving the similarity values. Though, it can be said that the performance of *imgSeek* would be acceptable for our purpose as the perceived latencies have been very low when performing the manual evaluation.

The library *LIRe* used in the theoretical approach in [55] for detecting phishing pages has been tested by using the demo application that is available for download. We tried to find matches for screenshots of HTTPS websites within the stored set of HTTP screenshots. Even for websites with not only one dynamic element, e.g. *http://en.wikipedia.org/*, the library did not yield any results. After cropping the screenshots to a maximum height of 500 pixels, the demo application found matching images. Unfortunately, it was irrelevant which image has been used as input, the search results kept the same, thus this library is not suitable for our purpose and has not been evaluated further.

As mentioned before, the *LevenshteinMatcher* that considers the whole HTML file has not been evaluated on the training set due to its infeasible runtime. Though, we measured the runtime on two average-sized webpages. According to a large-scale analysis of over 480,000 websites performed by HTTP Archive[1] the average size of an HTML document is about 60 KiB. Therefore, we generated two files of that size containing random characters and numbers (using `/dev/urandom` and `tr`) and took it as input for the *LevenshteinMatcher*. The distance calculation lasted longer than 37 seconds, hence far too long for large-scale evaluations.

For the rest of the algorithms, all similarity values calculated in the 50 test runs have been averaged per matcher and candidate pair. This yields the mean classification value of a matcher for every website pair in the training set. As for each candidate pair in the training set it is known if it is a positive or a negative match, it is possible to average the similarity values of all negative candidate pairs and all positive candidate pairs per matcher. Hence, these values denote the mean similarities the matchers calculate for positive and for negative candidate pairs. It is assumed that the broader the distance between those values is, the better the classification works.

The results of the matcher evaluation can be seen in Table 6.1. The second (AVG neg) and third column (AVG pos) of the table show the average values the corresponding matcher produces when checking a candidate pair that is a negative respectively a positive match. The fourth column (Mean) shows the arithmetic mean between these two values, i.e. $\frac{AVG\,0 + AVG\,1}{2}$. This value has been chosen as threshold for candidate pair classification of the corresponding matcher. The fifth column (Distance) is defined as the difference between AVG 1 and AVG 0 and serves as an indicator for robustness, thus it is used as selection criterion for the large-scale evaluation. The last column (Duration) shows the average time needed to calculate a similarity value of two webpages. The table is sorted by our main selection criterion for matchers, i.e. the distance. As it can be seen, the *LevenshteinWordMatcher* achieves the highest score in our evaluation followed by the *LevenshteinTextMatcher* and the *PathSimilarityMatcher*. The calculations of the *SSDeepImageMatcher* take an average of over 16 seconds for a single calculation which is infeasible for large-scale evaluations.

It makes sense not to use only one matcher for classification since either the semantic text or the structural elements would be ignored. We did not choose the two matchers with the highest scores, namely the *LevenshteinWordMatcher* in combination with the *LevenshteinTextMatcher* as both solely consider textual content. Hence, we selected a combination of the *LevenshteinWordMatcher* and the *PathSimilarityMatcher* for the large-scale evaluation since it is assumed that this yields the most accurate and robust results since both textual and structural elements will be considered. Moreover, the average duration of the *LevenshteinTextMatcher* shows a relatively high value giving another reason not to utilize this matcher.

Table 6.2 depicts a more precise breakdown of the test runs. The minimum, maximum and average calculated similarity values are shown for each category of the training set. When taking a look to the category *positive* it can be seen that the *WeightedTagSimilarity-*

[1]`http://httparchive.org/trends.php`, retrieved: 2014-11-21

Matcher	**AVG neg**	**AVG pos**	**Mean (Threshold)**	**Distance**	**AVG Duration**
LevenshteinWord	0.09	0.90	0.49	0.81	151
LevenshteinText	0.22	0.91	0.57	0.68	3358
PathSimilarity	0.37	0.99	0.68	0.62	48
LevenshteinCode	0.40	0.95	0.67	0.55	205
LevenshteinCodeWET	0.43	0.95	0.69	0.53	1577
SSDeepCode	0.32	0.80	0.56	0.48	118
SSDeepText	0.29	0.77	0.53	0.48	60
PQGram	0.06	0.50	0.28	0.43	291
SDHash	0.30	0.73	0.52	0.42	561
WeightedTagSimilarity	0.57	0.99	0.78	0.42	32
SSDeepImage	0.27	0.66	0.47	0.39	16399
imgSeek	0.07	0.40	0.23	0.34	-
SSDeep	0.26	0.59	0.42	0.33	70
FuzzyImage	0.66	0.91	0.78	0.25	247

Table 6.1: Results of the matcher evaluation

Matcher's average similarity value for positive candidates is, as well as the minimum, very high, thus having only little variations when encountering a HTTP-HTTPS equivalent. However, the calculations for *similar but negative* candidates vary between 0.02 and 0.98 reducing the robustness of this algorithm. Generally, most of the matchers show high variations for candidates in this category. The best values could be achieved by the *LevenshteinWordMatcher* which calculations vary between 0.02 and 0.38. The same matcher also shows very promising results in the category containing *negative* candidate pairs as well as the *PQGramMatcher* that achieved equal values.

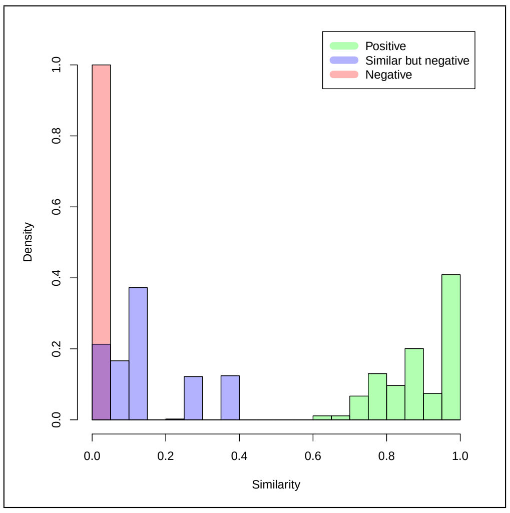
In Figure 6.3 the calculated similarity values for the websites in the training set are depicted as bar charts for some selected matchers. The chart of the *LevenshteinWord-Matcher* (see Figure 6.3a) shows some noticeable features. Firstly, the bars representing the *positive* category do not overlap with the bars of the other two categories, thus the threshold may be set within a certain range such that there is not a single wrong classification in the training set. The previously chosen threshold of the *LevenshteinWordMatcher* is within this range (see Table 6.3). Furthermore, it can be seen that all candidates of the *negative* category got minimal similarity values since they all are included in the first bin. The *LevenshteinTextMatcher* (see Figure 6.3b) shows almost as good results. Solely one bin of the *similar but negative* category overlaps with one bar of the *positive* candidate pairs. On closer inspection it turned out that this is caused by a candidate pair containing websites that both show the same legal text. The *PathSimilarityMatcher* (see Figure 6.3c) is able to distinguish candidates of the *positive* and *negative* categories very well but has difficulties with the *similar but negative* category. The algorithm calculated very high similarity values for some candidate pairs belonging to this group. The fourth

Matcher	**positive**			**similar but neg.**			**negative**		
	min	**max**	**avg**	**min**	**max**	**avg**	**min**	**max**	**avg**
LevenshteinWord	0.60	1.00	0.90	0.02	0.38	0.14	0.00	0.02	0.01
LevenshteinText	0.67	1.00	0.91	0.13	0.76	0.31	0.04	0.22	0.09
PathSimilarity	0.81	1.00	0.99	0.01	0.96	0.57	0.01	0.11	0.04
LevenshteinCode	0.50	1.00	0.95	0.02	0.88	0.54	0.08	0.42	0.18
LevenshteinCodeWET	0.50	1.00	0.95	0.02	0.89	0.57	0.09	0.48	0.21
SSDeepCode	0.40	1.00	0.80	0.00	0.83	0.48	0.00	0.99	0.08
SSDeepText	0.44	1.00	0.77	0.00	0.60	0.42	0.00	0.54	0.08
PQGram	0.03	1.00	0.50	0.00	0.63	0.10	0.00	0.02	0.01
SDHash	0.51	1.00	0.73	0.07	0.84	0.41	0.00	0.75	0.14
WeightedTagSimilarity	0.93	1.00	0.99	0.02	0.98	0.73	0.17	0.62	0.32
SSDeepImage	0.38	1.00	0.66	0.00	0.55	0.39	0.00	0.54	0.09
SSDeep	0.08	1.00	0.59	0.00	0.72	0.32	0.00	0.54	0.16
FuzzyImage	0.66	1.00	0.91	0.11	1.00	0.72	0.25	0.77	0.56

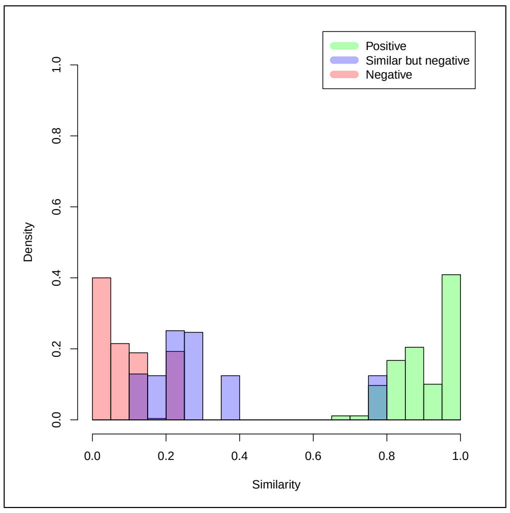
Table 6.2: Results of the matcher evaluation per training set category

depicted matcher (see Figure 6.3d) shows rather poor results since sometimes even bars of all three categories overlap. Therefore, the *FuzzyImageMatcher* is not suitable for providing reliable similarity matching for websites.

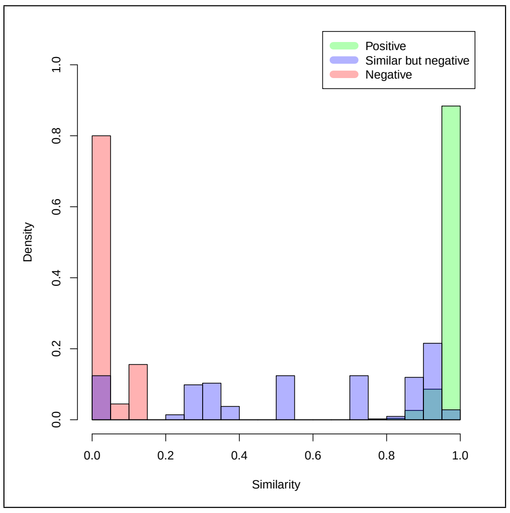
After we determined the threshold for each matcher by using the described method, we re-evaluated the data gathered while performing the test run on the training set and calculated the true/false positive (TP/FP) and true/false negative (TN/FN) values. With using these values, we computed precision and recall for each matcher which can be seen in Table 6.3. Furthermore, the percentage of correctly classified candidates is shown. Most of the matchers show a high recall score, thus the majority of the positive candidate pairs are also classified as positive matches. However, precision varies widely among the different matchers. As it can be seen, the *LevenshteinWordMatcher* classifies all of the candidate pairs in the training set correctly when using the calculated threshold. The *LevenshteinCodeMatcher* and the *LevenshteinCodeMatcherWithEndTags* comprise equivalent accuracy while the latter shows over seven times higher time consumption compared to the former (see Table 6.1). This suggests that considering HTML end-tags is not crucial for similarity matching based on structural elements.



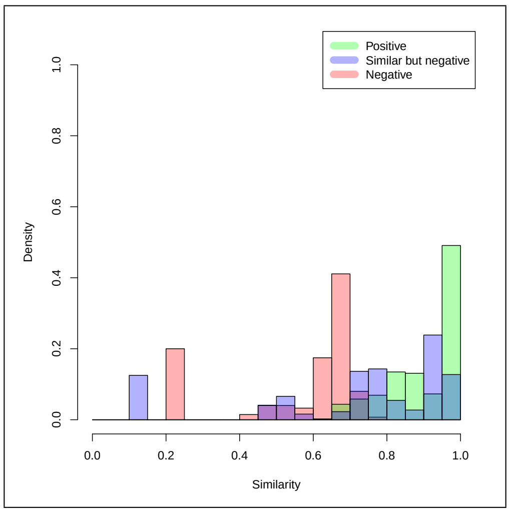
(a) LevenshteinWordMatcher



(b) LevenshteinTextMatcher



(c) PathSimilarityMatcher



(d) FuzzyImageMatcher

Figure 6.3: Similarity calculation results

Matcher	**classified correctly**	**precision**	**recall**
LevenshteinWord	1.00	1.00	1.00
LevenshteinText	0.95	0.84	1.00
PathSimilarity	0.78	0.56	1.00
LevenshteinCode	0.78	0.63	0.99
LevenshteinCodeWithEndTags	0.78	0.63	0.99
SSDeepCode	0.70	0.48	0.76
SSDeepText	0.83	0.82	0.93
PQGram	0.85	0.77	0.65
SDHash	0.79	0.78	0.99
WeightedTagSimilartiy	0.69	0.56	1.00
SSDeepImage	0.67	0.52	0.90
SSDeep	0.71	0.50	0.86
FuzzyImage	0.74	0.52	0.87

Table 6.3: Analysis of precision and recall

CHAPTER 7

Large-Scale Evaluation and Results

For the large-scale evaluation we chose to use Alexa's top million websites[1] (downloaded on 2014-11-13) as input for the *Candidate Generator*. This list is created by evaluating data that originate from over 25,000 different browser extensions [3] delivering data to Alexa. The rank of a website is determined by the estimated number of daily unique visitors and the estimated page visits of the past three months. However, it is stated [2] that websites with a rank beyond 100,000 are statistically not meaningful because of a lack of received data. Moreover, the list does not embrace subdomains of websites unless it is known that personal user pages (e.g. blogs) are hosted there [2].

In order to get an insight into the different TLDs occurring in the list we implemented a Python script and parsed the entries by using the package *publicsuffix*. The ten most frequent TLDs occurring in the Alexa top million list are stated in Table 7.1. As it can be seen, over 60% of the contained websites are hosted on the country independent `.com`-, `.net`- and `.org` domains. The most frequently occurring ccTLDs are `.ru` followed by `.de`.

The crawl has been performed using four distinct servers. Five instances of the *Candidate Generator* ran on an *Amazon EC2 T2 micro instance*[2]. Further three servers have been provided by SBA-Research and served as *Candidate Checkers*, whereby two of them ran five worker threads each and the remaining one ran 15. One of them additionally hosted the RabbitMQ server and the PostgreSQL server. Hence, the *Aggregator*, that accesses both of them, has also been deployed on that host.

After the first crawl of Alexa's top million pages it turned out that our software only considered about 706,000 distinct websites. The dropout of nearly 300,000 websites could

[1]Daily updated list of top million websites: `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`, retrieved: 2015-03-25

[2]`https://aws.amazon.com/ec2/instance-types/`, retrieved: 2015-03-13

TLD	Amount	%
.com	516927	51.7
.net	51912	5.2
.ru	42020	4.2
.org	41382	4.1
.de	27111	2.7
.jp	19362	1.9
.br	18984	1.9
.uk	17086	1.7
.in	15577	1.6
.pl	13544	1.4

Table 7.1: Most frequently occurring TLDs in Alexa list

have two reasons. Either the websites do not have an HTTPS equivalent with a valid certificate or the website is even unreachable via HTTP. In order to make sure we missed no webpages, we restarted the software with all unconsidered pages as input and with an extended connect timeout value of ten seconds instead of three. As a result of this re-run, an additional amount of 3,591 websites have been checked resulting in further 2,555 single rules.

The results of the crawl including the re-run allowed the generation of 128,539 rule files that contain 189,676 single rewriting rules for 181,689 different domains. Figure 7.1 shows the amount of input-domains that allowed the generation of at least one rule with respect to the rank in the Alexa list. That means that e.g. at least 19,256 of the websites with a rank lower than 100,000 do support HTTPS.

Figure 7.2 shows the calculated similarity values of both chosen matchers in the form of histograms.

Additionally, we evaluated the number of automatic redirects to HTTPS that occurred during the large-scale crawl. In total there were 53,382 automatic redirects. The ten subdomains that have been most frequently target of redirects to HTTPS can be seen in Table 7.2. Moreover, we evaluated the number of automatic redirects with respect to the corresponding Alexa rank. The results can be seen in Figure 7.3. Table 7.3 shows the same data with regard to the TLDs. In particular, we evaluated the TLDs of websites that provided automatic redirects as well as the TLDs of all generated rules. The ten TLDs which contained the most websites that allowed the generation of at least one rule are listed. The relative number in the table shows the number of input-domains of the Alexa list in relation to the corresponding absolute amount (e.g. 6.08% of the `.com`-domains included in the Alexa top million list performed an automatic redirect).

All of these results will be discussed in the next chapter.

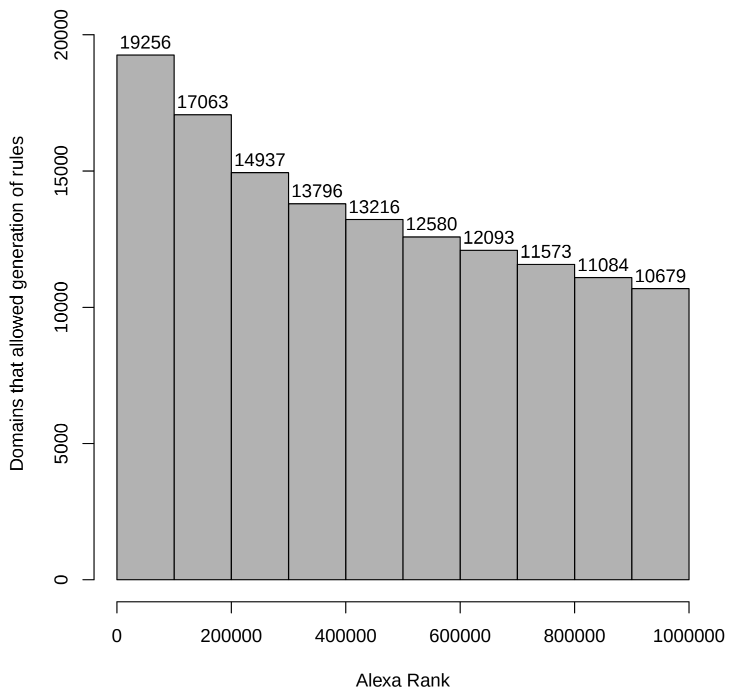
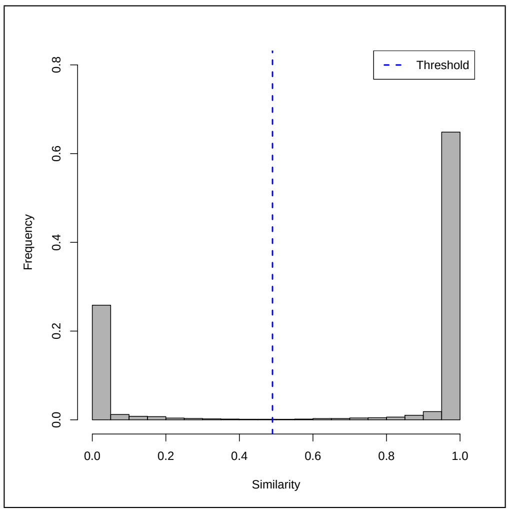
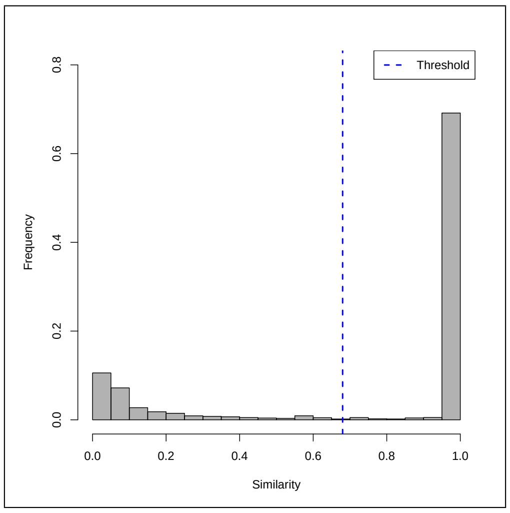


Figure 7.1: Number of generated rules



(a) LevenshteinWordMatcher



(b) PathSimilarityMatcher

Figure 7.2: Calculated similarity values

Subdomain	Amount	%
www.	31828	59.6
no subdomain	15789	29.6
accounts.	309	0.6
bankingportal.	168	0.3
secure.	164	0.3
account.	129	0.2
login.	95	0.2
portal.	82	0.2
ssl0.	78	0.1
banking.	73	0.1

Table 7.2: Automatic redirect evaluation

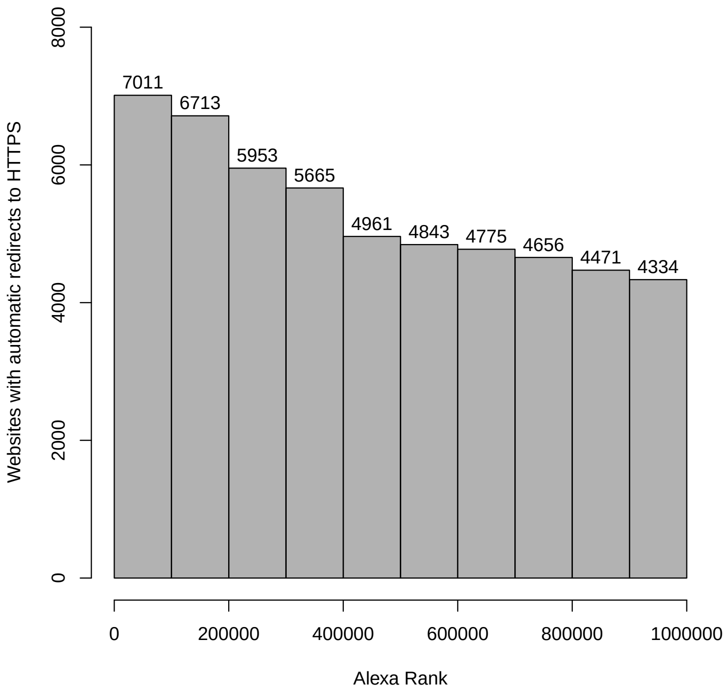


Figure 7.3: Number of automatic redirects to HTTPS

	Automatic Redirects		**Overall Results**	
TLD	**Absolute**	**Relative (%)**	**Absolute**	**Relative (%)**
com	31323	6.08	79284	15.39
org	2647	6.39	6672	16.10
de	2483	9.14	5794	21.32
net	1827	3.5	5399	10.35
co.uk	957	6.46	2986	20.15
com.au	597	7.79	1873	24.45
jp	526	4.69	1728	15.42
com.br	579	3.62	1727	10.79
ru	583	1.39	1672	3.99
nl	867	10.48	1602	19.36

Table 7.3: Results listed per TLD

CHAPTER 8

Discussion

As it can be seen in Figure 7.1 a higher amount of websites with low ranks in the Alexa list allowed the generation of rules than websites with higher ranks. These results suggest the assumption that well-known websites support HTTPS more likely than fameless ones.

Figure 7.2 shows that the classification of candidates has been clear most of the time since there are only a few calculated similarities near the threshold. Especially the *LevenshteinWordMatcher* calculated similarities either under 0.05 or over 0.95 for over 90% of all inspected candidates, thus making clear decisions in most of the cases. The *PathSimilarityMatcher* also performed very well and classified nearly 80% of the candidate pairs into the before mentioned intervals.

The subdomains that are target of redirects very often stated in Table 7.2 show that automatic redirects target either the *www.*-subdomain or no subdomain in nearly 90% of the cases. Both of these subdomains have been used for generating candidates since they also occurred in the ten most frequently used subdomains in the current *HTTPS Everywhere* rule set (see Table 4.1). Additionally, the fifth most frequent subdomain *secure.* has also been used for candidate generation. The subdomains *bankingportal.* and *ssl0.* appeared to be very special, thus we took a closer look on the underlying data. *bankingportal.* turned out to be the standard secure subdomain of many different German *Sparkassen*-websites. The multiple occurrences of the subdomain *ssl0.* arised from websites probably possessed by the webhoster *OVH*[1] all redirect to the same domain.

As it can be seen in Table 7.3, the number of websites performing automatic redirects to HTTPS sinks with increasing Alexa rank. Therefore, more popular webpages provide this security-increasing mechanism more likely than fameless ones.

Table 7.3 shows that the most rules could be generated for websites in the `.com`-TLD. The same applies to the number of automatic redirects. Since `.com`-domains form the majority of the Alexa list, i.e. the input for candidate generation, this could have been expected. However, the relative numbers show that there are TLDs that comprise a higher

[1] `https://www.ovh.co.uk/`, retrieved: 2015-04-13

percentage of domains that perform automatic redirects to HTTPS, like the `.nl`-TLD whose websites redirect in over 10% of the cases. The `.ru`-TLD brings up the rear in both shown categories with only less than 2% of automatic redirects and fewer than 4% of websites that allowed the generation of rules. The highest percentages of the latter category could be reached by the `.com.au`- and `.de`-TLDs with nearly 25% respectively about 21%.

There are also TLDs which allowed the generation of rules for 100% of the domains, like `.td`, `.tg` and `.tr`, but since there is only one domain for each of these TLDs in the Alexa list we do not discuss these corner cases.

We are currently in conversation with the responsible persons of the *HTTPS Everywhere* extension in order to integrate the generated rules into the public rule repository.

In order to assess the overall classification accuracy we chose a random set of 100 candidates and inspected them manually. We browsed both URLs of each candidate and evaluated if it is a positive or negative match. In this way we could determine the number of true/false positives (TP/FP) and true/false negatives (TN/FN). These results can be seen in Table 8.1. The cells highlighted in green show combinations that led to a correct classification, whereas cells highlighted in red show the number of misclassified candidates. 95% of the candidates in the sample have been classified correctly. The remaining five percent have been classified false positive by both matchers. One single candidate has been correctly identified as positive match by the *LevenshteinWordMatcher* but as negative match by the *PathSimilarityMatcher*.

		PSM			
		TP	**TN**	**FP**	**FN**
LWM	**TP**	54	-	-	0
	TN	-	40	1	-
	FP	-	0	5	-
	FN	0	-	-	0

Table 8.1: Random sample of 100 candidates out of all results

There are 50 candidates that got similarity values close (+/- 0.05) to the thresholds of both of the corresponding Matchers. These results have also been inspected manually in order to determine TP/TN/FP/FN rates again. We needed to exclude three of the candidates since one of them was not reachable anymore and other two have been double checked by the software, thus were duplicates in the database. The results can be seen in Table 8.2. To sum up, it can be said that over 70% of the candidates that are border cases have been classified correctly.

When solely considering the results of the *LevenshteinWordMatcher* it can be seen that it performs better than the *PathSimilarityMatcher*. It classified about 70% of the checked candidates correctly whereas the *PathSimilarityMatcher* handled only about 49% correctly.

In the following, the before mentioned introduction of the cache (see Section 5.5) and problems that occurred while implementing the software are discussed.

		PSM			
		TP	TN	FP	FN
LWM	TP	7	-	-	4
	TN	-	10	12	-
	FP	-	4	3	-
	FN	2	-	-	5

Table 8.2: Sample of candidates with similarities near the thresholds

The large speed-up of the *Candidate Checker* achieved by the utilization of a cache can be explained by the avoidance of accessing websites which cause a load timeout multiple times. As there are websites causing load timeouts the impact of using a cache can get very high since multiple accesses to such pages are avoided. Moreover, the overall system is designed in a way that makes the impact of the cache even higher. The *Candidate Checker* requests the HTTP webpage of the candidate pair first and the HTTPS site afterwards. Mostly there are multiple candidate pairs showing the same HTTP website, but different HTTPS websites. Therefore, the HTTP website gets requested multiple times, but the browser has to reload it every time since there has been a different site loaded in between, i.e. the corresponding HTTPS websites.

Furthermore, we encountered some issues with Selenium that made the development and data acquisition more difficult. One problem was that there is no interface to check if a redirect occurred when accessing an URL. Therefore, the URL is manually tested for change after a website was accessed. Obviously, it would be more convenient to have a callback method within Selenium for such cases. Additionally, there is no possibility to check the validity of a website's certificate. There are options to allow or deny access to websites with invalid certificates, but no interface for requesting information about it. Therefore, we could not gather any data about how many websites would provide a TLS connection, but do not show a valid certificate.

Moreover, the stability of Selenium respectively PhantomJS caused some problems since the Java process sometimes loses connection to the PhantomJS instance. In this case, a new PhantomJS browser is instantiated and the *Candidate Checker* continues to process work packages. Unfortunately, the PhantomJS process does not terminate when losing connection to the Java process and keeps running. As memory would run full after time due to the unused PhantomJS instances, we implemented a watchdog that kills these orphan processes periodically.

8.1 Limitations

In order to perform the evaluation in acceptable time, the Alexa top million websites have been used as they are and no variations of the domains have been tried. Since some websites might are accessible on a subdomain that is not included in the list (e.g. *http://google.com/* is in the Alexa list, but *http://www.google.com/* is not), there could have possibly been additional rules generated if the subdomain of the input websites would have been varied.

Furthermore, the content of linked JavaScript and CSS files is not considered. Hence, different versions of the scripts could be utilized without being detected by our software. Therefore, it is possible that websites get modified by scripts depending on the used protocol resulting in two entirely different webpages comprising the same HTML source code. However, since there is no use case for such behavior, it is very unlikely that legitimate websites act in that way.

Another problem is that Selenium does not provide any possibility to check the response code delivered by the HTTP server. Therefore, the software tries to match error pages as well. Obviously, the similarity will be very high when comparing two equal error-pages with each other resulting in a rule for a website that is not available at all. The approach to probe the response code manually by using the standard Java HttpURLConnection also failed due to the fact that some websites perform redirects by using meta tags on their error pages although there are dedicated response codes for redirections (see Section 2.2).

8.2 Future Work

Currently the software solely calculates a similarity value for the landing page of a website. In order to make sure the whole website supports HTTPS it could be a desirable goal to parse the landing page for further URLs pointing at the same domain and compare their HTTP and HTTPS versions as well. This would make it possible to find exclusion patterns making the rule set more accurate.

Moreover, the generation of candidates could be improved by using the repository of the ZMap Team[2]. It contains port scans of every IPv4 address on the internet gathered with their tool ZMap [28]. The scan results could be used for finding hosts that provide both HTTP and HTTPS servers. Since the corresponding domain names have to be known in order to perform a correct HTTP request, they have to be found in some way. One opportunity is to use reverse DNS to resolve the IP addresses to domain names. However, not every IP has a corresponding PTR-record thus this method is not very reliable. Another possibility is given by inspecting certificates that are provided by the HTTPS servers. They contain the corresponding domain name they are issued for and are also available in the repository of the ZMap Team.

[2]`https://scans.io/`, retrieved: 2015-03-16

CHAPTER 9

Conclusion

In this work we evaluated the current rule set of the *HTTPS Everywhere* extension and extracted the most frequently used subdomains for HTTPS secured websites. Most of the rewriting rules redirect to the base domain, the `www` subdomain and the same subdomain the HTTP URL showed. Furthermore, we evaluated 15 different methods for measuring the similarity of websites. We found that the best matchers for websites were the *LevenshteinWordMatcher*, the *LevenshteinTextMatcher* and the *PathSimilarityMatcher* since they had the largest distance between their mean classification for positive and negative matches. In total we generated about 190,000 single rules for nearly 129,000 different domains and are in the process of submitting them to the public *HTTPS Everywhere* rule set.

Appendix A - Training Set

	Website A	Website B
Positive	http://www.gmx.at/	https://www.gmx.at/
	http://www.youtube.com/	https://www.youtube.com/
	http://www.amazon.de/	https://www.amazon.de/
	http://www.amazon.co.jp/	https://www.amazon.co.jp/
	http://en.wikipedia.org/wiki/Main_Page	https://en.wikipedia.org/wiki/Main_Page
Similar but negative	http://members.chello.at/wienerrettung/m70/Vorschriften/SanG.htm	https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=20001744&ShowPrintPreview=True
	https://github.com/openpreserve/bitwiser/blob/master/bitwiser-core/src/main/java/eu/scape_project/bitwiser/utils/SSDeep.java	https://github.com/openpreserve/bitwiser/blob/master/bitwiser-core/src/main/java/eu/scape_project/bitwiser/BitwiseAnalyser.java
	https://www.google.at/?gws_rd=ssl#q=a	https://www.google.at/?gws_rd=ssl#q=b
	http://en.wikipedia.org/wiki/Fuzzy_logic	http://en.wikipedia.org/wiki/Fuzzy_set
	http://www.amazon.co.jp/	https://www.amazon.de/
	http://www.amazon.com/	https://www.amazon.co.jp/
	http://www.amazon.com/	https://www.amazon.de/
	http://www.amazon.com/Instant-Video/b/ref=nav_shopall_aiv?ie=UTF8&node=2858778011	https://www.amazon.de/Instant-Video/b/ref=nav_shopall_aiv?ie=UTF8&node=3010075031
Negative	http://www.amazon.com/	https://www.youtube.com/
	http://www.asdf.com/	https://scholar.google.at/
	http://www.gmx.at/	https://www.facebook.com/
	http://en.wikipedia.org/wiki/Main_Page	https://www.google.com/
	http://derstandard.at/	https://translate.google.at/

Bibliography

[1] W. Alcorn, C. Frichot, and M. Orrù. *The Browser Hacker's Handbook.* John Wiley and Sons Inc., 2014.

[2] Alexa. *How are Alexa's traffic rankings determined?* `https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined-`, retrieved 2015-04-10.

[3] Alexa. *Information. Insight. Advantage.* `http://www.alexa.com/about`, retrieved 2015-04-09.

[4] N. Augsten, M. Böhlen, and J. Gamper. Approximate Matching of Hierarchical Data Using Pq-grams. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 301–312. VLDB Endowment, 2005.

[5] A. Barth. *HTTP State Management Mechanism*, Apr. 2011. `http://tools.ietf.org/html/rfc6265`, retrieved: 2014-12-12.

[6] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol version 2 (draft)*, July 2014. `https://tools.ietf.org/html/draft-ietf-httpbis-http2-14`, retrieved: 2014-12-11.

[7] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, May 1996. `http://tools.ietf.org/html/rfc1945`, retrieved: 2014-12-11.

[8] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[9] I. A. Board. *IAB Statement on Internet Confidentiality*, 2014. `https://www.iab.org/2014/11/14/iab-statement-on-internet-confidentiality/`, retrieved: 2015-01-08.

[10] F. Breitinger, K. Astebø l, H. Baier, and C. Busch. mvHash-B - A New Approach for Similarity Preserving Hashing. In *IT Security Incident Management and IT Forensics (IMF), 2013 Seventh International Conference on*, pages 33–44, Mar. 2013.

[11] F. Breitinger and H. Baier. A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance. *7th annual Conference on Digital Forensics, Security and Law (ADFSL)*, pages 89–101, May 2012.

[12] F. Breitinger and H. Baier. Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2. In M. Rogers and K. Seigfried-Spellar, editors, *Digital Forensics and Cyber Crime*, volume 114 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 167–182. Springer Berlin Heidelberg, 2013.

[13] F. Breitinger, G. Ziroff, S. Lange, and H. Baier. Similarity Hashing Based on Levenshtein Distances. In G. Peterson and S. Shenoi, editors, *Advances in Digital Forensics X*, volume 433 of *IFIP Advances in Information and Communication Technology*, pages 133–147. Springer Berlin Heidelberg, 2014.

[14] A. Z. Broder. Identifying and Filtering Near-Duplicate Documents. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, COM '00, pages 1–10, London, UK, UK, 2000. Springer-Verlag.

[15] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic Clustering of the Web. In *Selected Papers from the Sixth International Conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[16] D. Buttler. A Short Survey of Document Structure Similarity Algorithms. In *International Conference on Internet Computing*, pages 3–9. CSREA Press, 2004.

[17] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[18] K.-T. Chen, J.-Y. Chen, C.-R. Huang, and C.-S. Chen. Fighting Phishing with Discriminative Keypoint Features. *Internet Computing, IEEE*, 13(3):56–63, May 2009.

[19] M. Chew. *Firefox 32 supports Public Key Pinning*, Aug. 2014. `http://monica-at-mozilla.blogspot.de/2014/08/firefox-32-supports-public-key-pinning.html`, retrieved 2015-03-25.

[20] S. Cobb. *New Harris poll shows NSA revelations impact online shopping, banking, and more*, Apr. 2014. `http://www.welivesecurity.com/2014/04/02/harris-poll-nsa-revelations-impact-online-shopping-banking/`, retrieved: 2015-03-02.

[21] M. Corporation. *Regular Expressions*, Feb. 2015. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions`, retrieved: 2015-02-09.

[22] R. Dhamija, J. D. Tygar, and M. Hearst. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.

[23] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*, Jan. 1999. `https://www.ietf.org/rfc/rfc2246.txt`, retrieved: 2015-02-10.

[24] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1*, Apr. 2006. `https://tools.ietf.org/html/rfc4346`, retrieved: 2015-02-10.

[25] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*, Aug. 2008. `https://tools.ietf.org/html/rfc5246`, retrieved: 2015-02-10.

[26] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*, Apr. 2014. `https://tools.ietf.org/html/draft-ietf-tls-rfc5246-bis-00`, retrieved: 2014-11-27.

[27] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[28] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 605–620, Washington, D.C., 2013. USENIX.

[29] EFF. *HTTPS Everywhere Rulesets*. `https://www.eff.org/de/https-everywhere/rulesets`, retrieved: 2015-02-09.

[30] K. S. Enforcer. *FAQ*. `https://code.google.com/p/kbsslenforcer/wiki/FAQ`, retrieved: 2015-02-10.

[31] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, June 1999. `http://tools.ietf.org/html/rfc2616`, retrieved: 2014-12-11.

[32] S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese. Detecting Structural Similarities between XML Documents. In *Proceedings of the 5th International Workshop on the Web and Databases*, WebDB '02, 2002.

[33] S. Flesca, G. Manco, E. Masciari, L. Pontieri, and A. Pugliese. Fast detection of XML structural similarity. *Knowledge and Data Engineering, IEEE Transactions on*, 17(2):160–175, Feb. 2005.

[34] Fox-IT. *DigiNotar Certificate Authority breach "Operation Black Tulip"*, Sept. 2011. `http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/`

2011/09/05/diginotar-public-report-version-1/rapport-fox-it-operation-black-tulip-v1-0.pdf, retrieved: 2015-03-02.

[35] A. Fu, L. Wenyin, and X. Deng. Detecting Phishing Web Pages with Visual Similarity Assessment Based on Earth Mover's Distance (EMD). *Dependable and Secure Computing, IEEE Transactions on*, 3(4):301–311, Oct. 2006.

[36] M. Green. *On the new Snowden documents*, Dec. 2014. http://blog.cryptographyengineering.com/2014/12/on-new-snowden-documents.html?utm_source=dlvr.it&utm_medium=twitter&m=1, retrieved: 2015-03-02.

[37] R. W. Hamming. Error detecting and error correcting codes. *BELL SYSTEM TECHNICAL JOURNAL*, 29(2):147–160, 1950.

[38] M. Hara, A. Yamada, and Y. Miyake. Visual similarity-based phishing detection without victim site information. In *Computational Intelligence in Cyber Security, 2009. CICS '09. IEEE Symposium on*, pages 30–36, Mar. 2009.

[39] M. Henzinger. Finding Near-duplicate Web Pages: A Large-scale Evaluation of Algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 284–291, New York, NY, USA, 2006. ACM.

[40] A. Hlaoui and S. Wang. A new algorithm for inexact graph matching. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 4, pages 180–183 vol.4, 2002.

[41] J. Hodges, C. Jackson, and A. Barth. *HTTP Strict Transport Security (HSTS)*, Nov. 2012. http://tools.ietf.org/html/rfc6797, retrieved: 2014-11-04.

[42] C.-R. Huang, C.-S. Chen, and P.-C. Chung. Contrast Context histogram-An Efficient Discriminating Local Descriptor for Object Recognition and Image Matching. *Pattern Recogn.*, 41(10):3071–3077, Oct. 2008.

[43] M. K. Kern and E. Phetteplace. Hardening the browser. *Reference & User Services Quarterly*, 51(3):210–214, 2012.

[44] M. Khonji, Y. Iraqi, and A. Jones. Phishing Detection: A Literature Survey. *Communications Surveys Tutorials, IEEE*, 15(4):2091–2121, Apr. 2013.

[45] J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digit. Investig.*, 3:91–97, Sept. 2006.

[46] M. Kranch and J. Bonneau. Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning. In *NDSS '15: The 2015 Network and Distributed System Security Symposium*, Feb. 2015.

[47] N. Krawetz. *Looks Like It.* http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html, retrieved: 2015-02-18.

[48] A. Langley. *Overclocking SSL*, June 2010. https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html, retrieved: 2015-03-04.

[49] A. Langley. *Public Key Pinning*, May 2011. https://www.imperialviolet.org/2011/05/04/pinning.html, retrieved 2015-03-25.

[50] L. Leitão, P. Calado, and M. Weis. Structure-based Inference of XML Similarity for Fuzzy Duplicate Detection. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 293–302, New York, NY, USA, 2007. ACM.

[51] W. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[52] M. Lux and S. A. Chatzichristofis. LIRe: Lucene Image Retrieval: An Extensible Java CBIR Library. In *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, pages 1085–1088, New York, NY, USA, 2008. ACM.

[53] M. Marlinspike. New tricks for defeating SSL in practice. *BlackHat DC, February*, 2009.

[54] M. Marlinspike. *SSL And The Future Of Authenticity*, Aug. 2011. https://www.youtube.com/watch?v=pDmj_xe7EIQ, retrieved 2015-03-09.

[55] M.-E. Maurer and D. Herzner. Using Visual Website Similarity for Phishing Detection and Reporting. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 1625–1630, New York, NY, USA, 2012. ACM.

[56] S. Media. *New level of Internet domain names - more choice for NZ-ers*, 2013. http://www.scoop.co.nz/stories/SC1310/S00023/new-level-of-internet-domain-names-more-choice-for-nzers.htm, retrieved: 2015-03-05.

[57] E. Medvet, E. Kirda, and C. Kruegel. Visual-similarity-based Phishing Detection. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 22:1–22:6, New York, NY, USA, 2008. ACM.

[58] B. Möller, T. Duong, and K. Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback, 2014.

[59] Nominet. *.uk roll out*, 2014. http://www.dotuklaunch.uk/dates-and-definitions, retrieved: 2015-03-05.

[60] M. Pawlik and N. Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. *Proc. VLDB Endow.*, 5(4):334–345, Dec. 2011.

[61] H. Perl, S. Fahl, and M. Smith. You Won't Be Needing These Any More: On Removing Unused Certificates from Trust Stores. In N. Christin and R. Safavi-Naini, editors, *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 307–315. Springer Berlin Heidelberg, 2014.

[62] M. Perry. *TorFlow: Tor Network Analysis*, Aug. 2009. `http://fscked.org/talks/TorFlow-HotPETS-final.pdf`, retrieved: 2015-03-12.

[63] E. Rescorla. *HTTP Over TLS*, May 2000. `https://tools.ietf.org/html/rfc2818`, retrieved: 2014-12-02.

[64] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*, Jan. 2012. `https://tools.ietf.org/html/rfc6347`, retrieved: 2015-02-10.

[65] I. Ristić. *Bulletproof SSL and TLS*. Feisty Duck, 2014.

[66] S. Roopak and T. Thomas. A Novel Phishing Page Detection Mechanism Using HTML Source Code Comparison and Cosine Similarity. In *Advances in Computing and Communications (ICACC), 2014 Fourth International Conference on*, pages 167–170, Aug. 2014.

[67] V. Roussev. Data Fingerprinting with Similarity Digests. In K.-P. Chow and S. Shenoi, editors, *Advances in Digital Forensics VI*, volume 337 of *IFIP Advances in Information and Communication Technology*, pages 207–226. Springer Berlin Heidelberg, 2010.

[68] V. Roussev, G. G. Richard, III, and L. Marziale. Multi-resolution Similarity Hashing. *Digit. Investig.*, 4:105–113, Sept. 2007.

[69] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The Emperor's New Security Indicators. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 51–65, May 2007.

[70] C. Soghoian and S. Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL (Short Paper). In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, FC'11, pages 250–259, Berlin, Heidelberg, 2012. Springer-Verlag.

[71] R. S. Stanković and B. J. Falkowski. The haar wavelet transform: its status and achievements. *Computers & Electrical Engineering*, 29(1):25 – 44, 2003.

[72] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 399–416, Berkeley, CA, USA, 2009. USENIX Association.

[73] Y. Takama and N. Mitsuhashi. Visual similarity comparison for Web page retrieval. In *Web Intelligence, 2005. Proceedings. The 2005 IEEE/WIC/ACM International Conference on*, pages 301–304, Sept. 2005.

[74] M. Theobald, J. Siddharth, and A. Paepcke. SpotSigs: Robust and Efficient Near Duplicate Detection in Large Web Collections. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 563–570, New York, NY, USA, 2008. ACM.

[75] A. Tridgell. *spamsum*, 2002. `http://www.samba.org/ftp/unpacked/junkcode/spamsum/README`, retrieved: 2014-12-16.

[76] W3C. *The Original HTTP as defined in 1991*, 1991. `http://www.w3.org/Protocols/HTTP/AsImplemented.html`, retrieved: 2015-03-17.

[77] W3C. *HTML 4.01 Specification*, Dec. 1999. `http://www.w3.org/TR/1999/REC-html401-19991224/`, retrieved 2015-03-17.

[78] W3C. *HTML5 specification*, Oct. 2014. `http://www.w3.org/TR/2014/REC-html5-20141028/`, retrieved 2015-03-17.

[79] w3schools. *JavaScript RegExp Reference*. `http://www.w3schools.com/jsref/jsref_obj_regexp.asp`, retrieved: 2015-02-09.

[80] M. Weis and F. Naumann. DogmatiX Tracks Down Duplicates in XML. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 431–442, New York, NY, USA, 2005. ACM.

[81] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style Host Authentication with Multi-path Probing. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 321–334, Berkeley, CA, USA, 2008. USENIX Association.

[82] Wikipedia. *Algorithm Implementation/Strings/Levenshtein distance*. `https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#Java`, retrieved: 2014-11-3.

[83] H. Zhang, G. Liu, T. Chow, and W. Liu. Textual and Visual Content-Based Anti-Phishing: A Bayesian Approach. *Neural Networks, IEEE Transactions on*, 22(10):1532–1546, Oct. 2011.

[84] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: A Content-based Approach to Detecting Phishing Web Sites. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 639–648, New York, NY, USA, 2007. ACM.