# Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Andrej Staruch

**Advisor:** RNDr. Marek Kumpošt, Ph.D.

# Abstract

The goal of this master's thesis is to design and implement a system, which will compute potential risk for a given URL. The computation of potential risk is based on extendable series of individual test suites, and the result of weighted tests is a number called 'phishing score'. Based on this number, the application can automatically allow or block the given communication. Optionally, the end user could be warned and decide if he wants to proceed to this website.

## Keywords

phishing detection, Phishtank, machine learning classification, docker, anti-phishing, URL testing, Trusted Network Solutions

# Contents

# 1 Introduction

In today's world, the threat of a cyber attack cannot be ignored. There are a plethora of companies that try to protect their customers from potential damage such as: getting infected by a virus, getting ransomware or malware, getting stolen their business intelligence.

This thesis is concerned with another part of the cybercrime called phishing. Phishing is a social engineering attack to obtain sensitive information such as user names, passwords, credit card details, bank account credentials for malicious reasons. Because of a massive attack vector, there is not a reliable way or a tool to prevent such an attack consistently in every business sector.

This thesis reviews several ways how to detect a phishing attack, designs and implements system for checking whether a given URL is phishing, and provides a generic interface for easy integration to other tools that can use it on the network perimeter (e.g., proxy, firewall).

Designing an anti-phishing system has similar characteristics as designing an anti-virus system - we are trying to protect users before malicious attempts of an attacker. However, as we can see, users are still infected when they are using anti-virus software, so designing a correct and powerful anti-phishing system is an uneasy task. This thesis focuses on the extendability of the anti-phishing system for future editions.

# 2 Theoretical background

What is phishing? Many academic papers or security companies have different definitions of phishing. We can look at several papers to have a broader look for this problem:

> Phishing is a trap where any targeted individual, is communicated by someone impersonating as a legitimate and a reputed organization to entice the individual into providing sensitive information such as banking information, credit card details, and passwords.
>
> (Certain Investigation on Web Application Security [5])

> Phishing is a social engineering attack that aims at exploiting the weakness found in system processes as caused by system users.
>
> (Phishing Detection: A Literature Survey [19])

> The technique used to perform on-line robbery/stealing of person credentials is called phishing in cyber international.
>
> (A secured methodology for anti-phishing [9])

For a more exhaustive definition, we can peek at Anti-Phishing World Group definition, that covers most of the previous definitions:

> Phishing is a criminal mechanism employing both social engineering and technical subterfuge to steal consumers' personal identity data and financial account credentials. Social engineering schemes use spoofed e-mails purporting to be from legitimate businesses and agencies, designed to lead consumers to counterfeit Web sites that trick recipients into divulging financial data such as usernames and passwords. Technical subterfuge schemes plant crimeware onto computers to steal credentials directly, often using systems to intercept consumers' account user names and passwords – and to corrupt local navigational infrastructures to misdirect consumers to counterfeit Web sites (or authentic Web sites through

> phisher-controlled proxies used to monitor and intercept consumers' keystrokes).
>
> (Phishing Activity Trends Report [54])

For this work, we will define phishing as a fraudulent attempt combining social engineering skills with a technical trickery to obtain sensitive data from other parties.

What is recent news saying about phishing? We have read several reports from security companies, and our findings are the following:

- In a survey of 808 IT security professionals from companies all over the world, 56% identified targeted phishing attacks as their biggest current cybersecurity threat (followup answers were 51% insider threats, 48% ransomware/malware). [11]
- Out of 41,686 reported security incidents, 2,103 were data breaches. One third (32%) of them involved phishing, which was the highest number in this category (the following are stolen credit cards and keyloggers). An interesting fact is that only 2.99% of users clicked on the link during phishing exercises (previous years were 5%, 7%, 9%). [1]
- Phishing attempts have grown by 65% from the previous year. [12]

The predictions for the near future are that phishing will grow more, so this is a problem that needs to be addressed more.

## 2.1 Phishing types

In this section, we will look at how different types of phishing attacks are developed.

Phishing attacks are usually divided into several categories based on the who is the target:

1. Deceptive phishing - this is the most common attack where an attacker is trying to steal money or credentials from the victim, usually done by sending a fake e-mail from a bank with a fake URL link, where account details are exposed to the attacker.

This attack is typically made in batches on a large group of victims, typically on a leaked database with e-mails.

2. Spear-phishing - this attack involves more social engineering skills and is targeted on single units instead of a wide group like in deceptive phishing. Attackers need to perform detailed research of its victim, making it difficult to mark this attack as fraudulent.

3. Whaling - similar to spear phishing, but attackers take a considerable time to prepare the attack and usually target executive officers because they have more privileges and knowledge than a common employer.

Since the attacker needs more preparation, spear-phishing and whaling work more with the social engineering techniques [17]. The typical situation is that an attacker will get as much information as possible, and then through an e-mail (2.1) conversation will make the victim send money.

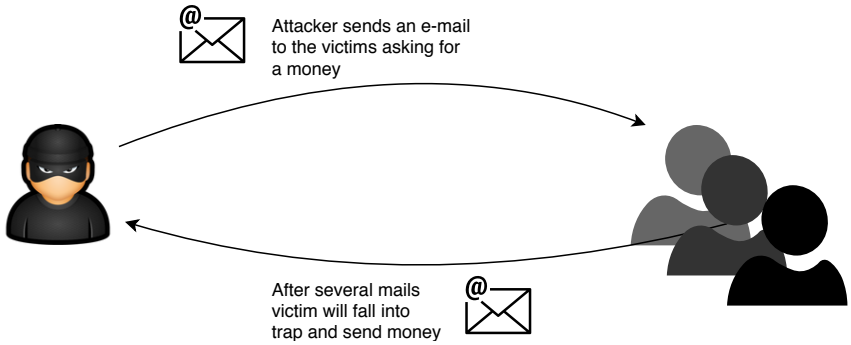


Figure 2.1: Simple e-mail demonstration

The other common situation is that an attacker starts a phishing campaign (figure 2.2), which usually does not last for a long time, and sends many e-mails to the compromised addresses (which were usually obtained during some security breach of a company).

If a user is already on a fake site, while he believes it is a correct one, it is usually late for him, and an attacker gets his credentials. The

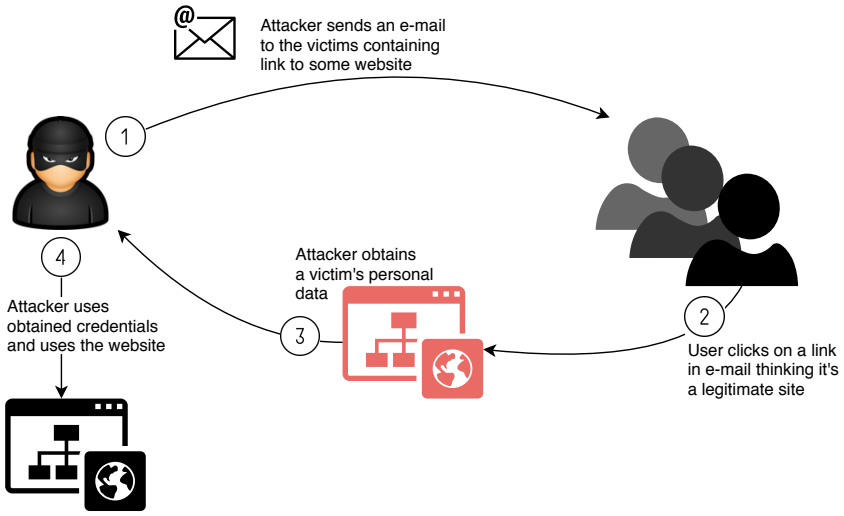


Figure 2.2: Phishing campaign

defense system should prevent him from visiting such a site. This thesis will be addressing such a problem.

To this date, there have been numerous papers published and research done about phishing and how to prevent it. We will look thoroughly at them and propose a system that can protect a user from phishing sites. The targeted audience is a user using a device such as a company proxy or a firewall on a network layer. We can see where our system will fall on a figure 2.3.

## 2.2 Current defense mechanisms against phishing

In this section, we will look at how can the user reduce the risk of being defrauded.

### User education

One of the best defense is user awareness and education on the phishing topic [3]. This usually consists of several actions which may be done in an organization:

- create a security team which will handle reported incidents

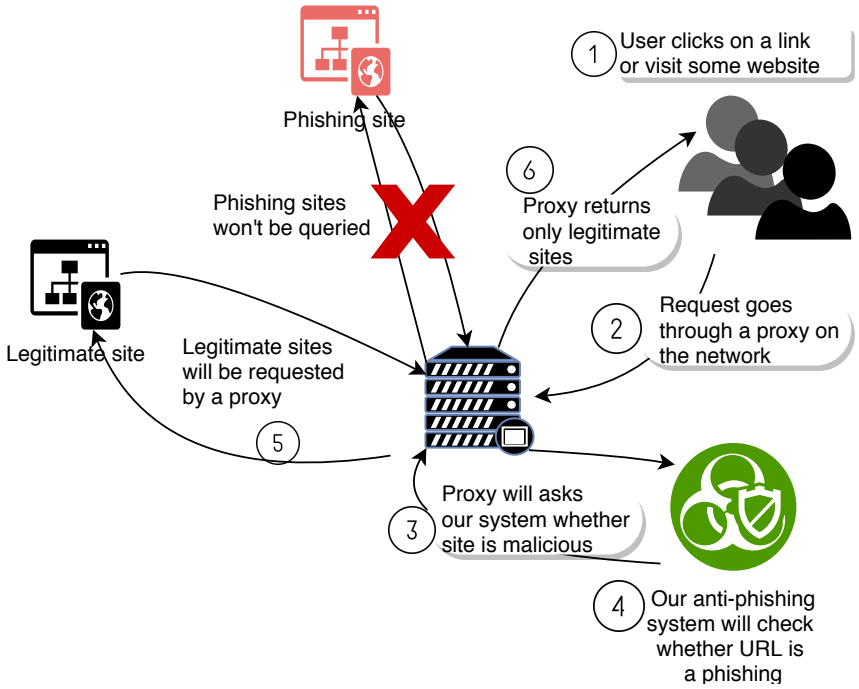


Figure 2.3: Our system in network

- explain how usually the phishing attack looks like and how to notice clues that this e-mail may be a phishing attempt
- periodically send information about the current trends and phishing techniques from the organization's security team
- remind about the best practices on how to prevent fraudulent attempts
- hire an external team that will make a controlled attack and then provide a report with guidelines on what to do in the future differently in order to prevent such a situation

### Threat-detection applications

If the user falls into the trap and clicks on the link from a phishing e-mail, there is another protective layer, which can prevent visiting a

site: a modern browser. All of the current modern desktop browsers (Chrome, Firefox, Opera, Safari, Edge) and mobile browsers (Chrome, Safari, UC Browser, Samsung Internet, Opera, Android) has incorporated some basic protection against vicious sites. This can be checking if the visited site is on some blacklist, or if it contains a virus.

Another application that is helping to prevent visiting a phishing (or malicious site) is an anti-virus. We have checked well-known anti-virus software and found that ESET [1], Avast [2] and Kaspersky [3] has a specific module for phishing detection.

## 2.3 Phishing trends

We have previously mentioned that phishing attacks and campaigns are still on the rise, now we will look more on recent data so that we can frame the problem. The Anti-Phishing Working Group [4] has been founded in 2003, and since then, it has been a collaboration between 2200+ global institutions to counteract and response to cybercrime. It is the biggest consortium facing the phishing problem.

### Volume of attacks

The APWG is publishing quarterly phishing reports since the year 2012. They are available online and contain the current trends for a relevant time period with numeric metrics for the following things:

- The number of unique phishing websites. "The APWG tracks the Web sites across the globe. In their statistics, a single phishing site may be advertised as thousands of customized URLs (all leading to the same attack destination)."
- The number of phishing (e-mail) campaigns. "The APWG tracks the number of unique phishing reports it receives from peers. An e-mail campaign is a unique e-mail sent out to multiple users, directing them to a specific phishing web site (multiple campaigns may point to the same web site). APWG counts

1. https://www.eset.com/us/anti-phishing/
2. https://blog.avast.com/avast-improves-phishing-detection-avast
3. https://www.kaspersky.com/resource-center/threats/spam-phishing

unique phishing report e-mails as those found in a given month that have the same e-mail subject line".

- The number of brands targeted by phishing campaigns.

We have compiled the data from the reports published during the period since October 2012 till the June 2019 (publications [31]–[54]) and put them on a graph (figure 2.4 on page 10). We can see that the graph contains spikes, which are probably the outcome of changing the way of how to gather the data or some big partner is joining or leaving the consortium. We can see from this graph that phishing is still a big problem. In table 2.1, we can see total numbers for the provided period. We need to point out that these are only the numbers that are recorded from the companies in APWG – real number for the world wide web is estimated to be much higher.

Table 2.1: E-mail campaigns, websites and brands statistics

| | E-mails campaigns | Websites | Brands |
|---|---|---|---|
| Total | 6,400,431 | 5,026,891 | |
| Monthly (median) | 69,925 | 51,232 | 358 |
| Monthly (average) | 79,018 | 62,060 | 358 |

Data source: APWG 2012/10 – 2019/06

### Length of a phishing campaign

Sheng et al., 2009 [58] have discovered that the average phishing campaign lasts around 2 hours (the time between 1st and last phishing e-mail) whereas the time needed for detecting and taking down the website is much larger. Detailed statistics are in the table 2.2.

As this data is quite old, we have decided to do a research to find out how long take it will take now to take down a site now. For this purpose, we will use a site called Phishtank. Phishtank [55] is an online collaborative tool to track and share phishing data. Anyone can submit fraudulent URLs, and users then manually verify whether the website is malicious or not.

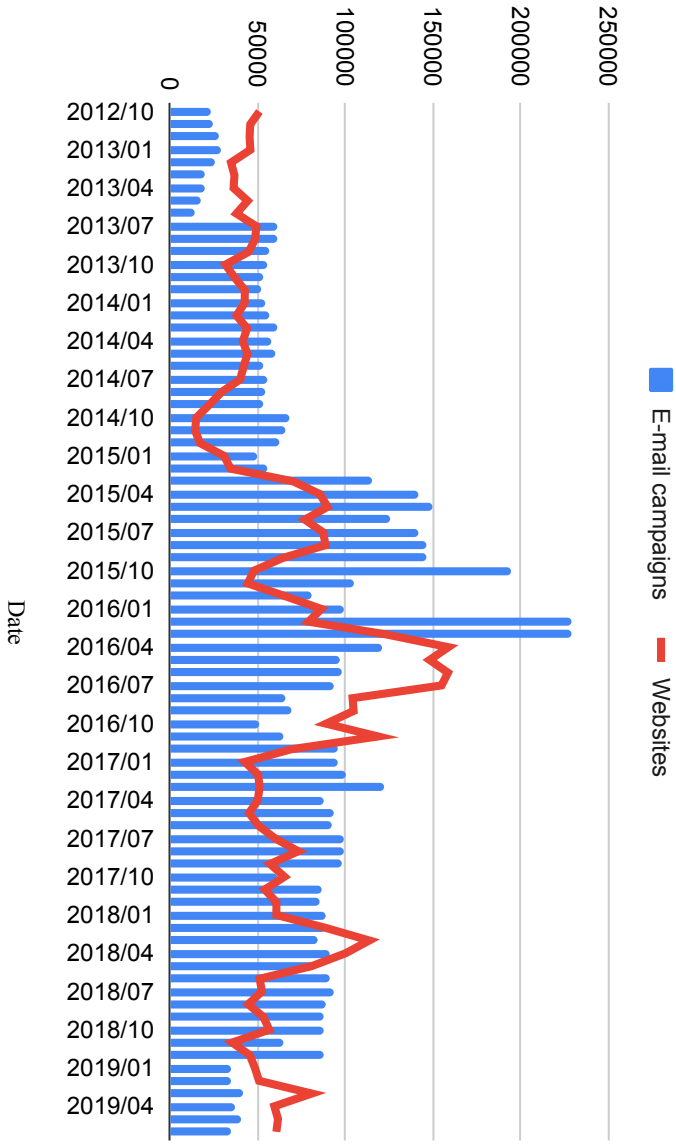


Figure 2.4: Phishing campaigns and websites trends

Table 2.2: Longevity of phishing sites in 2009 [58]

| Time elapsed (hours) | Websites taken down (%) | Phishing campaigns finished (%) |
|---|---|---|
| 0 | 2.10% | 0% |
| 2 | 7.90% | 63% |
| 4 | 17.80% | 67% |
| 12 | 33.00% | 72% |
| 24 | 57.60% | 75% |
| 48 | 72.30% | 90% |

The service provides public API for obtaining feed with the websites, which are classified as phishing and are also online. As we do not want to pollute this part with the technical details, we would like to refer a reader to the section 5.3, where we have described how we made a program, that periodically downloads Phishtank database and updates our database with the data.

We have gathered Phishtank data from November 2018 till June 2019. The total number of submitted and confirmed phishing websites during this 10-month long period is 119,043. The average monthly number (11,904) is five times smaller than what we can see in APWG reports during the last few years. We should notice that Phishtank is relying on user submissions, whereas APWG consists of a large number of organizations. We have put the numbers in figure 2.5 over the same period. We can see that phishing has an increasing trend.

The second attribute that we have looked at how long does it take to take down a site. Since we have queried Phishtank once per day for the data, we do not know precisely the time when the site went down. Our data provide only the information for how long is a website online after it is classified as phishing. We can see that almost half of the sites are still online after one week. That means that even when the site is classified as a phishing website, the site is still accessible. Our system should prevent users from visiting such a site.

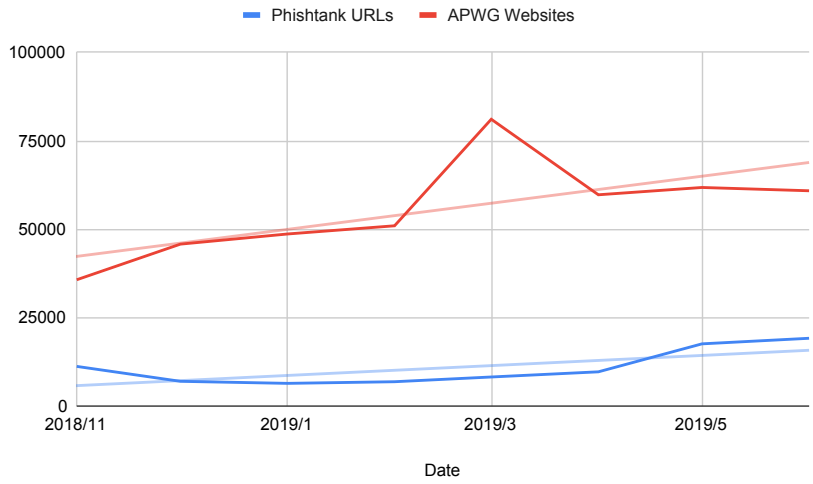


Figure 2.5: APWG vs Phishtank stats over 2018/11 - 2019/06

Table 2.3: Time to take URL down

| Time elapsed | Websites taken down | Time elapsed | Websites taken down |
|---|---|---|---|
| 0 days | 2.9% | A week | 54.6% |
| 1 day | 19.5% | A month | 79.8% |
| 2 days | 31.2% | A year | 95.5% |

Data source: Phishtank 2018/11 – 2019/08

# 3 Detection methods

As we now have awareness about what is phishing and how the attack may look, we should now dive into how we can protect a user from such behavior. We can usually categorize techniques for evaluating if the site is phishing or not into several categories:

- checking the site against blacklists or public lists
- URL analysis
- HTML/content analysis
- visual analysis
- behavioral analysis

The anti-phishing system is then usually a combination of several different approaches together to mitigate the cons of one category with a combination of another. We will introduce each category separately, their pros and cons, and evaluate real data, whether we should use it in our system.

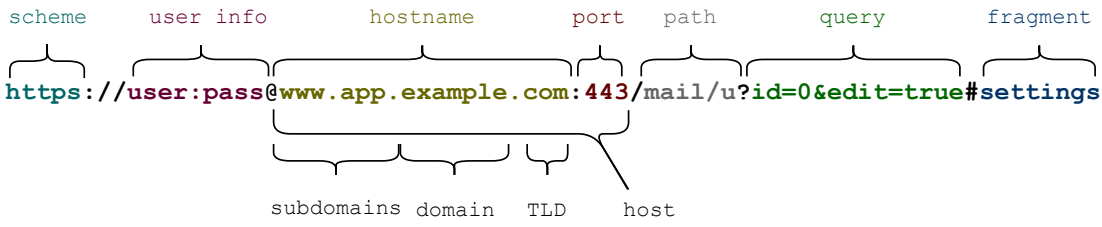


Figure 3.1: URL structure

## URL data sets

In this work, we will heavily work with a Uniform Resource Locator (URL) [6]. URL is an identifier, which tells the resource location on a network. The whole URL is usually composed of mandatory parts like

`scheme`, `hostname`, and other complements, which we can see in the figure 3.1. More detailed info about URL can be found in the RFC3986.

To have a bigger picture picture of how the various features may impact phishing detection, we will need to evaluate it on some data. For this purpose, we have prepared several data sets from different sources containing various URLs.

**Kernun**[1] data set contains URLs that were processed by the Kernun firewall on the 2018/11/1 in a Czech company. The original number of URLs is 6,568,187. We have processed this list of URLs and filtered only URLs that are still online and then randomly selected URLs to contain only one unique hostname. The final size is 73,082.

**Phishtank** data set was built by aggregating daily data for a 13-months long period (Sep. 2018 – Oct. 2019). This data set contains 163,891 URLs, which were classified as phishing by the Phishtank service. We have filtered it, so the final set contains unique hostnames. The final size is 83,303. Since most of these URLs have been taken down, we have downloaded a new fresh list from 13/11/2019 and reduced it to contain only live URLs and unique hostnames. The number of records was reduced to 9,820 URLs.

**UNB** data sets are publicly available on the University of New Brunswick, Canada [63]. They have created various datasets for various purposes (spamming, phishing or just regular traffic) and also published benign data set which was built by obtaining more than 35,000 hostnames from the Alexa[2] and then used a crawler to extract URLs from these hostnames, filtered it for unique URLs and then ran through VirusTotal[3] to filter malicious URLs. The size of original data set was 35,378. We have filtered it, so the final list contains only live URLs without duplicate hostname. The final size of the filtered set is 267 records.

**Openphish** data sets were built from Openphish feed, which was initially 2,797 phishing URLs classified as phishing by Openphish service. We have filtered it to contains only unique hostnames. The final number of URLs is 1,573.

1. Kernun is the name for a family of products from the company Trusted Network Solutions which is a partner of this thesis
2. Alexa is site which is publishing top visited sites in the world
3. VirusTotal is service providing checking if URL contains malicious file or if given file is a virus

These data sets were used for two different purposes: firstly for analysis on a large number of URLs, how phishing URL and benign URL look like, then filtered datasets were used for training of detection model.

With the help of these data sets, we should be able to perform estimations concerning the following features.

## 3.1 List based approach

This approach is based on publicly known lists operated either by some authority or a community. We usually perform two operations with these lists - whitelisting and blacklisting. There is no difference between malicious categories in the general lists – we can't say that this URL contains malware, but is not phishing. However, some services provide only phishing info.

The advantage of a list-based approach is that we have a binary information about the inspected URL - either the site is malicious or it is not. The disadvantage is that it takes some time from the discovery of a malicious URL until the moment when the URL is listed. There is some delay, and in many situations, this is not what we want.

We will now introduce several public services, that can help us with our problem.

### Phishtank

Phishtank [55] is an online collaborative tool to track and share phishing data. Anyone can submit fraudulent URLs, users then verify if a given website is a phishing site or not. Phishtank provides a public API where a client can perform a lookup for one URL through the POST request [4] or he can request a current snapshot of the live (websites are not taken down) database [5]. We are already using a Phishing database for data exploration, but we can also check whether a URL is phishing one.

```
$ curl -X POST \
  'https://checkurl.phishtank.com/checkurl/index.php?url=https://www.example.org/' \
  -H 'User-Agent: phishtank/thesis-phishing-detection'
>
```

4. https://www.phishtank.com/api_info.php
5. https://www.phishtank.com/developer_info.php

```
...
<response>
    <url0><url><![CDATA[https://www.example.org/]]></url>
        <in_database>false</in_database>
    </url0>
</response>
```

### OpenPhish

OpenPhish [26] is automated platform for phishing intelligence. It analyses given URLs from various partners, and if a URL is a phishing one, it provides a variety of metadata and information. Service is not providing an API that we can use, but they are providing a list with live phishing URLs, which we can cache beforehand and check it when we need it on our side. It is similar to PhishTank in the way that they are both providing URL feeds.

### Google Safe Browsing

Google provides it's own lists through a Safe Browsing API [16]. The client application can query the API and get information if it has any threat type (malware, social engineering, or other). This API is included in many applications, such as Google Chrome, Mozilla Firefox, and many other products. We will also use this as one of the relevant sources.

### Domain Name Service (DNS) lists

The two following lists are a little bit different than the previous three. They are working with DNS-records instead of URLs. DNS[24] is a hierarchical naming system used for domain name resolving. It is an equivalent of the phone book, but for domains. Every public website is somewhere hosted physically and it has been assigned a public IP address. Humans are bad at remembering lots of numeric addresses, so there exists an authority which is translating IP addresses to hostnames. For example the hostname `www.muni.cz` is translated to an IP address `147.251.5.237`. The DNS record may have several types. We will be interested mostly in `A` and `AAAA`, which are used for IPv4 and IPv6, respectively. If there is no record, then the response code `NXDOMAIN` is returned. Tools that we can use for DNS lookups are `dig` or `nslookup`. Here is an example of for simple DNS lookup:

```
$ nslookup muni.cz
Non-authoritative answer:
Name: muni.cz
Address: 147.251.5.239
$ nslookup non-existing-hostname.cz
** server can't find non-existing-hostname.cz: NXDOMAIN
```

#### DNS-based Blackhole List

DNSBL [65] is used for real-time stopping of the spam. Spam campaigns are usually going together with phishing campaigns so we may leverage this fact in our system. Lookup in DNSBL is performed by requesting DNS `A` record for reversed octets of the IP address followed by a blacklist provider. If the record is returned, then the host is blacklisted in this specific list. We can then obtain more info through `TXT` record. If we have received `NXDOMAIN` return code, then the IP address is not listed in *this* specific list. However, there is a large list of DNSBL providers, so it is a good idea to perform a lookup to many of them. Here is an example of a lookup for an IP address `139.162.99.243` from the DNSBL provider `https://www.usenix.org.uk/content/rbl.html`:

```
$ nslookup 243.99.162.139.all.s5h.net
Non-authoritative answer:
Name: 243.99.162.139.all.s5h.net
Address: 127.0.0.2
```

Since the DNS record is found, the IP address is used for spamming.

#### DNS-based White List

DNSWL [66] has taken, on the contrary, an another approach - they are gathering IP addresses and hosts, which are consistent and have a low percentage of spam. This option is mainly for future extensions because blacklisting is not possible for many hosts in the IPv6 universe. We will not use this method in our system for now.

## 3.2 URL analysis

Another possible approach is the analysis of a URL. We can look at the properties of the URL and say with the help of several rules, whether

this URL looks like phishing or not. Take as an example these two URLs: `http://paypal.com` and `http://my.paypal-com.submission.flightkaart.com/onodera/webapps/881a1/websrc`.

The first one is a public service Paypal, the second one has Paypal in the URL, but the domain is *flightkaart*. The site is trying to convince the user that it has something to do with Paypal, but in fact, it's hosted somewhere else and there is a high chance that this is a phishing site. Also it is using a lot of subdomains, which is less user-friendly, and again this small thing may signify that this is a phishing site.

Throughout the following paragraphs, we will introduce several heuristic features that may or may not help us detect whether it is a phishing URL. We will then transform these features into numeric representations, so we can then apply machine learning algorithms to find out, which of these features are relevant for us.

The advantage of URL analysis is that we can perform it in real-time – we do not need to wait for network responses, so the reaction time is much lower than in the list-based methods. The disadvantage is that we can only *guess* whether the URL is malicious because this is a heuristic approach.

We have read and studied several academic works that are concerned with this topic. During the following subsection, we will introduce several URL features that look promising or other groups of authors successfully used them in their systems. We will also introduce new features that we think may be useful and we did not see any other work which uses them.

We will work with our previously defined data sets. For each feature, we will define a formula for computing a normalized value (value between 0 and 1) that we will later use in a machine learning algorithms.

#### Usage of the IP address [25] [28] [59]

We have introduced DNS in section 3.1. To set up DNS records, we have to own a domain. Domain registrars charge money for selling domains. To minimize the cost of a phishing attack, the attacker may omit the process of setting up the domain and use an IP address directly. This behavior is suspicious for the end-user and there is a chance that the website is phishing.

The IP address could also be encoded into other formats, such as binary, octal, decimal, and hexadecimal. So our first feature will be checking whether an URL is in a numerical format and we will give it the following value:

$$v_1 = \begin{cases} 1 & \text{if hostname is IP address;} \\ 0 & \text{otherwise.} \end{cases}$$

Here are examples of the URLs which would have value *1* for this feature.

**Feature: IP address in hostname**

```
http://138.197.136.185/National%20Bank%20Online.html
https://0254.0331.027.0356
http://0x308f647/-credit-agricole-france/
https://2899908590
```

We have looked into our data sets and found out that the IP address is almost equally represented in both data sets:

Table 3.1: IP address in hostname

| | IP address in hostname | Ratio |
|---|---|---|
| Phishtank | 1,274 | 1.46% |
| Kernun | 1,163 | 1.59% |

### Length of an URL [59]

The length of the URL is a common practice to hide the doubtful part from the user. There are several papers that are saying that the length above some threshold is looking suspicious ([8] [21] [20] [22]). This metric is one of the best classifiers for machine learning algorithms according to [22].

Other studies also state that we can use all parts from the URL such as the sub-domain, domain, path, query, and the path token count.

We can see in table 3.2 that Kernun data set does not contain any record which has a URL fragment. The original data set with more

Table 3.2: Length of the URL and its parts

| | Kernun | | | Phishtank | | |
|---|---|---|---|---|---|---|
| Percentile | 50% | 75% | Max | 50% | 75% | Max |
| URL | 55 | 92 | 402 | 50 | 74 | 4173 |
| Hostname | 18 | 23 | 77 | 19 | 25 | 249 |
| Path | 14 | 27 | 386 | 19 | 37 | 843 |
| Query | 50 | 116 | 384 | 38 | 119 | 4094 |
| Fragment | N/A | N/A | N/A | 23 | 32 | 3819 |

than 6 million records contain only 5 URLs with the URL fragment. The low number of records with the fragment is probably because the system is not processing them.

This information about URL parts length will help us to set the constants for the following formula for computing feature value:

$$v_2 = \frac{\text{length}_{\text{URL}} - \text{length}_{\text{min URL}}}{\text{length}_{\text{max URL}} - \text{length}_{\text{min URL}}}$$

The value $v_2$ will represent where on the range from min to max will fall the specific length.

#### `user_info` [25] [56] [59]

URL can have a symbol @ used for login information. Nowadays, this is not used because of security issues, but malicious attackers can try to obfuscate the phishing URL because a majority of users check only the start of a URL. The following example is actually pointing to `2782399.azureedge.net`, which is obviously wrong for the user.

**Feature: Symbol @**

`https://support.microsoft.com@2782399.azureedge.net`

We will say that URL is phishing if it contains credentials:

$$v_3 = \begin{cases} 1 & \text{if the hostname contains user info;} \\ 0 & \text{otherwise.} \end{cases}$$

However, our finding is that there are only 2 and 7 URLs in Kernun and Phishtank data sets, respectively. Several facts could cause these low numbers:

- it is easily detectable, so phishers are not bothering to use this technique
- links are resolved and the information with `user_info` is lost

**Nested sub-domains [25] [56] [28] [59]**

The technique of nesting sub-domains is similar a approach like with the long URLs - trying to mask a host with a targeted brand. Academic works mention that 4 sub-domains are considered as phishing. On the other hand, lot of websites now use Amazon webservices (or other cloud services) for hosting their legitimate sites and their URL can looks like `https://ec2-18-223-122-56.us-east-2.compute.amazonaws.com/`. Also the nested subdomains are usually used for internal systems with signing service. In our datasets, we found a following distribution:

Table 3.3: Distribution of subdomains

| Nr. of subdomains | 0 | 1 | 2 | 3 | 4+ |
|---|---|---|---|---|---|
| Kernun | 18.0% | 64.9% | 16.3% | 0.6% | 0.2% |
| Phishtank | 46.5% | 42.1% | 7.8% | 2.3% | 0.7% |

From this data, we will say that the URL is classified as phishing if it contains more than two subdomains:

$$v_4 = \begin{cases} 1 & \text{if it has more than 2 subdomains;} \\ 0 & \text{otherwise.} \end{cases}$$

Following URL will be classified as phishing for this value:

**Feature: Nested sub-domains**

`http://history.meps.tp.edu.tw/sites/hstl/`

**Usage of HTTPS [29] [25]**

For a long time the ratio of HTTP vs HTTPS web sites was in favor for the unencrypted sites. Then the company Let's Encrypt started providing a free HTTPS certificate, which could be obtained through an API and the overall look of the World Wide Web had changed. We can see in the figure 3.2 that in the year 2017 there was for the first time more HTTPS sites than HTTP and since then the number rapidly grew in favor of HTTPS. In our data-sets the numbers are following:

Table 3.4: Ratio of HTTPS sites in dataset

| HTTPS sites | Count | Ratio |
|---|---|---|
| Kernun | 29,016 | 39.70% |
| Phishtank | 36,243 | 41.51% |

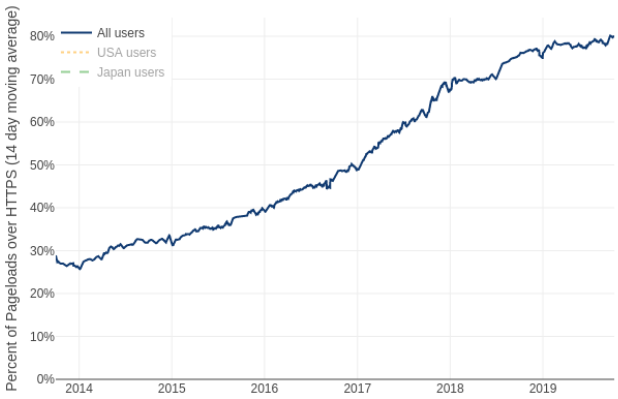


Figure 3.2: Percentage of Web Pages loaded by Firefox using HTTPS. Source: Firefox

We will set our feature that if `https` is missing, it's a phishing site:

$$v_5 = \begin{cases} 0 & \text{if the URL uses } \texttt{https}; \\ 1 & \text{otherwise.} \end{cases}$$

### Extra `https` token [59]

If `https` certificate is not acquired legitimately, phishers may try to convince the user that an extra word token secures the site. There is an assumption that lots of users do not pay attention to the hierarchy of a URL, and when they see a string `https`, they think that the site is secure. In our datasets, there were 66 (0.08%) URLs in the Phishtank data set and 0 in Kernun.

The value for this feature is one if a hostname starting with a `https` is phishing.

$$v_6 = \begin{cases} 1 & \text{if the URL contains extra } \texttt{https} \text{ token;} \\ 0 & \text{otherwise.} \end{cases}$$

Example of an URL in our data set which has this property:

**Feature: Extra `https` token**

`http://www.https-myetherwallet.net/`

### Shortening service [59] [23]

Shortening service is a method of transforming a relatively long URL into a shorter one, which can user memorize or send over services, which are limiting the length of a message or make it smaller for other reasons. Technically this is done over HTTP redirect method 301 - shortening service server will redirect a client to the original URL.

Phishers may try to confuse a user with legitimate services (`bitly.com`, `goo.gl`) to initially obtain a feeling that the link is a normal one. When users are already on a website, they may not see that the URL is now long or suspicious.

In our data set, there were 14 URLs using shortening in the Kernun data set, and 27 in Phishtank. However, as we have deleted URLs with the same hostname, we cannot take these values since we have altered the data set. If we look into original data sets, we can find that Kernun contained 0.66% URLs with shortening service and Phishtank 2.19%.

$$v_7 = \begin{cases} 1 & \text{if the URL host is shortening service;} \\ 0 & \text{otherwise.} \end{cases}$$

Here is an example of a phishing URLs in our data set:

**Feature: Shortening service**

`https://tiny.cc/GnjUIz`

resolves to
`https://bankieren.rabobank.nl.scannerplus.be/nl/service/online-bankieren/nieuwe-rabo-scanner-aanvragen/bankpas_aanvragen.php`

### Non-standard port [59]

The standard ports are defined by IANA [6] organization.For `http` it is port 80, for `https` the port is 443. The most used port by phishing sites (except the standard one) is port number `81`. Our findings are that 0.35% sites from Kernun use a non-standard port, whereas 0.68% Phishing sites are using a non-standard port.

$$v_8 = \begin{cases} 1 & \text{if port of the URL is other than 80 or 443;} \\ 0 & \text{otherwise.} \end{cases}$$

Example from data set:

**Feature: Non-standard port**

`http://jppost-yo.com:81/dao.html`

### Special characters

Several works ([29], [18], [20], [22]) are describing that presence of the special characters such as '!', '@', '#', '$', '%' '^̂', '&', '*', '(', ')', '_', '-', '+', '/', '
', or '=' can be used as another feature. We have analysed total number of special characters in the URL parts. The findings were following:

6. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt

| | Kernun | | | Phishtank | | |
|---|---|---|---|---|---|---|
| | 50% | 99% | Max | 50% | 99% | Max |
| URL | 6 | 56 | 122 | 5 | 26 | 222 |
| Path | 2 | 16 | 120 | 3 | 14 | 220 |
| Query | 7 | 63 | 100 | 4 | 31 | 121 |
| Fragment | NA | NA | NA | 2 | 28 | 102 |

Based on these data, we will set constants for computing feature value for each component:

$$v_9 = \frac{\text{\# of spec. chars} - \text{min \# of spec. chars}}{\text{max \# of spec. chars} - \text{min \# of spec. chars}}$$

### Special keywords

Another metrics can be the number of special keywords that are present in phishing URLs. The work CANTINA+ [67] states that words "secure", "account", "webscr", "login", "ebayisapi", "signin", "banking", "confirm" are sensitive ones. We will check for these words in URL and compute value as following:

$$v_{10} = \frac{\text{\# of spec. keywords} - \text{min \# of spec. keywords}}{\text{max \# of spec. keywords} - \text{min \# of spec. keywords}}$$

### Global Top Level Domain (TLD)

Another feature will look at the top-level domain. There are 7 global TLDs: `com`, `org`, `net`, `int`, `edu`, `gov`, `mil`. We will also include our local domains `cz` and `sk`. The value of this feature is as follows:

$$v_{11} = \begin{cases} 1 & \text{if the TLD is not from global TLD or cz/sk;} \\ 0 & \text{otherwise.} \end{cases}$$

Kernun data set contains 87.73% URLs from the mentioned TLDs, Phishtank only 58.89%.

#### Extra www prefix

Some phishing URLs may try to mimic the targeted brand by adding a `www` prefix. There are 0.04% URLs in the Kernun dataset and 0.08% in the Phishtank dataset with this feature.

$$v_{12} = \begin{cases} 1 & \text{if the hostname contains extra } \texttt{www-}; \\ 0 & \text{otherwise.} \end{cases}$$

**Feature: Extra `www` prefix**

`www-paypal-com-s.vpn.bua.edu.cn`

#### More than 4 consecutive digits [20]

The last feature from the URL analysis is a check if the hostname contains more than four consecutive digits. In our data sets, Kernun has 0.60% sof uch URLs, Phishtank has 0.62%.

$$v_{13} = \begin{cases} 1 & \text{if the hostname has 4 consecutive digits;} \\ 0 & \text{otherwise.} \end{cases}$$

**Feature: Extra `www` prefix**

`http://8185415634156jay8.blob.core.windows.net`
`http://168000036458002.16mb.com/1000655841.html`

## 3.3 Analysis of an HTML

Previous features were based purely on the given URL and can be evaluated immediately. It doesn't matter, whether the URL has phishing content or not, we will classify it just on the static properties.

To counterfeit this property of the previous category, we will introduce features, which will analyze the HTML page, which can change over time.

Some features will have a constant for the computation of the feature value. This constant has been derived after we have analyzed our dataset[7].

The drawback of this method is that we need to wait until the page is downloaded. This adds a small overhead to decide whether the site is phishing.

#### Input field [25] [27] [56]

The main reason for existence of phishing sites is, that an attacker wants to obtain credentials from a victim. If there is not present an `<input>` field, there is less chance that attacker will get victim's credentials.

After analysis of our sets, we have found that a reasonable number of input tags is around 50. Our value for the feature will be computed as follows:

$$v_{14} = \frac{\text{number of } \texttt{input} \text{ fields}}{50}$$

#### `src` attribute [25] [56]

Several HTML tags may have `src` attribute. This attribute is used for sourcing images or other assets. Usually, the page has assets on the localhost and serves them by the use of relative URLs. If the phishing site is a simple copy of another popular site, then we can check whether the domain in URL and `src` attributes match.

$$v_{15} = \frac{\text{number of links pointing to another domain}}{\text{number of all links}}$$

We need to denote that there is a rise of third-party providers or so-called content delivery networks (CDN), which are hosting the content around the world so that the user can download it from a geographically better location.

#### Anchor element [29] [25]

Anchor element is an elementwith tag `<a>` and with its `href` attribute creates a hyperlink to local files, sources, e-mails or other URLs. We

7. Source: `./data-science/notebooks/html-features.ipynb`

will count ratio of links pointing to other domain against localhost.

$$v_{16} = \frac{\text{number of links pointing to another domain}}{\text{number of all links}}$$

### Form handler

This feature is examining submit form handler [8]. Legitimate sites usually have some meaningful action with a form-data. Phishing sites may use empty actions because they are not really interested in the followup action. We will check for actions with targets to `about:blank`, `blank`, `#skip`, `#`, `javascript:true` and count elements containing them. The value for this feature is as follows:

$$v_{17} = \frac{\text{number of suspicious form actions}}{5}$$

**Feature: Form handler**

```
<form action="about:blank" method="get"></form>
```

### Invisible `iframe`

This feature is looking for iframes with invisible borders - the attacker may try to insert an iframe over a valid site with input fields, sending the information to the phishing site. We will count the suspicious iframes and return normalized value:

$$v_{18} = \frac{\text{number of suspicious iframes}}{12}$$

### Rewritting statusbar [59] [25]

Attacker can change the URL address bar to something else with the use of `onMouseOver`. Example of this behaviour:

8. `https://www.w3schools.com/php/php_forms.asp`

**Feature: onMouseOver**

```
<a onMouseOver="window.status=https://bad-url.com"
href="..."/>
```

We will check for the presence of such an element:

$$v_{19} = \begin{cases} 1 & \text{if page can rewrite status bar;} \\ 0 & \text{otherwise.} \end{cases}$$

**Disabled right click on a page [29] [25] [59]**

Disabling right-click on page is considered as bad practice. Normal sites don't have a reason for blocking it. So therefore, there is consideration that blocking a right click is only on a malicious site. Example of disabling the right click:

**Feature: Disabled right-click**

```
<body oncontextmenu="return false;">
```

We will check for the presence of such an element and the value is as follows:

$$v_{20} = \begin{cases} 1 & \text{if page has disabled the right click;} \\ 0 & \text{otherwise.} \end{cases}$$

**PopUp Window [29] [25] [59] [2]**

It's uncommon to ask for a user credentials through a pop up window. Because of that, we are checking existing pop-up windows and computing value with following formula:

$$v_{21} = \frac{\text{number of popup windows}}{20}$$

**Favicon**

The favicon of a website increases its credibility. By default, the favicon is loaded from the local machine. We can look if a website has a favicon

and if it points to the current domain. If not, there is a chance that the site is copying foreign favicon.

**Feature: Favicon**

```
<link rel="shortcut icon"
href="https://example.com/myicon.ico">
```

$$v_{22} = \begin{cases} 1 & \text{if favicon is loaded from an external site;} \\ 0 & \text{otherwise.} \end{cases}$$

#### Modern framework

We can try to examine if phishers adapted to the new technologies and look for signs that are inclining to that.

$$v_{23} = \begin{cases} 0 & \text{if site is using Angular, React,} \\ & \text{Vue, Ember, Backbone or Meteor;} \\ 1 & \text{otherwise.} \end{cases}$$

#### Title containing hostname

Another feature that we can look at is whether the page title contains the hostname. Most of the legitimate services have in the page title, so if the phishing site is trying to convince the user that he is on some specific site, there will be a mismatch of the hostname and the title.

$$v_{24} = \begin{cases} 0 & \text{if title contains hostname;} \\ 1 & \text{otherwise.} \end{cases}$$

### 3.4 Visual analysis

The visual analysis consists of analysis how the page looks like. If the site is targetting some brand, we may compare screenshots of this site and conclude whether it is phishing or not.

Guang-Gang Geng et al. [14] propose that favicon could be used.

We will not use these techniques in our system since they are getting poor results compared to other methods.

## 3.5 Behavioral analysis

In this section, we look at the features that are not analyzing HTML nor URL, but usually analyze domain and its properties.

The disadvantage of these methods is that we are relying on the 3rd parties and we need to use network requests, which adds latency to our system.

### HTTP redirect

HTTP redirect can be used by an attacker to hide the malicious URL behind another URL. If a URL is redirected, the user may not notice the change and think that he is on the correct URL.

$$v_{25} = \begin{cases} 1 & \text{if page is using redirect;} \\ 0 & \text{otherwise.} \end{cases}$$

### Google index

Google indexes a page if the Google crawler has visited it. It is then analyzed for content and meaning, and then stored in the Google index. Indexed pages can be shown in Google Search results[9].

$$v_{26} = \begin{cases} 0 & \text{if URL is indexed by google;} \\ 1 & \text{otherwise.} \end{cases}$$

### DNS record

We can gain much of the useful information from the DNS record. If there is no existing DNS record for a website, we can consider it as a risk due to the reason that the attacker does not want to spend time

9. https://www.google.com/search/howsearchworks/crawling-indexing/

setting up it for a short campaign.

$$v_{27} = \begin{cases} 0 & \text{if site has DNS record;} \\ 1 & \text{otherwise.} \end{cases}$$

DNS is providing a security extension called DNSSEC. If the site has DNSSEC there is a higher chance that it is not a phishing one.

$$v_{28} = \begin{cases} 0 & \text{if site has configured DNSSEC;} \\ 1 & \text{otherwise.} \end{cases}$$

The last two things that we can check are when was the DNS record created and when it was updated. If the record was created a long time ago there is a high chance that the site is not new and thus is a valid site since many phishing sites are there only for a limited time. The same logic applies to the last update time. If the DNS record was updated recently, it could be a hint that the site is new.

$$v_{29} = \begin{cases} 1 & \text{if DNS record was updated recently;} \\ 0 & \text{otherwise.} \end{cases}$$

$$v_{30} = \begin{cases} 0 & \text{if DNS record was created 2 years ago and more;} \\ 1 & \text{otherwise.} \end{cases}$$

We are obtaining this info through a service `whois`.

#### SSL certificate

We can also obtain information from the SSL certificate used for securing the HTTP protocol. If the site has HTTPS protocol, we can inspect the certificate, its claimed authority, and for how long is the certificate valid.

Authority

$$v_{31} = \begin{cases} 0 & \text{if hostname is claimed authority in the certificate;} \\ 1 & \text{otherwise.} \end{cases}$$

Certificate dates

$$v_{32} = \begin{cases} 0 & \text{if SSL certificate is fresh;} \\ 1 & \text{otherwise.} \end{cases}$$

We will gather this info through a command-line client `openssl`.

### HTTP response headers

We can also look at the HTTP response headers. If the website is using modern HTTP response headers focused on security, there is a high chance that they are trying to protect a user from the data theft and so the site should be non-malicious.

### HTTP Strict Transport Security

The HTTP Strict-Transport-Security response header (HSTS) tells the client that it should only be accessed using HTTPS instead of HTTP. If there should be performed any communication over insecure HTTP, the connection is by the client (usually a browser) changed to HTTPS, or if there is no HTTPS possible, the connection is refused.

$$v_{33} = \begin{cases} 0 & \text{if HSTS header is implemented;} \\ 1 & \text{otherwise.} \end{cases}$$

### Cross site scripting protection

Cross-site scripting (XSS) is an attack from the family of injections. The attacker is trying to inject malicious script through an unsanitized input form. Another end-user then does not know that the script is not coming from the actual server, but from an attacker. The server can implement some level of protection by using the X-XSS-Protection header.

$$v_{34} = \begin{cases} 0 & \text{if XSS protection is implemented;} \\ 1 & \text{otherwise.} \end{cases}$$

### Content Security Policy

Content Security Policy (CSP) is another layer of security, which helps to detect and prevent attacks such XSS, data injection, or man in the middle.

CSP defines a set of rules that are locking down an application and are mitigating the risk of attacks that needs a resource to be embedded into the page.

$$v_{35} = \begin{cases} 0 & \text{if CSP rules are implemented;} \\ 1 & \text{otherwise.} \end{cases}$$

### X-Frame-Options

X-Frame-Options header has been developed to prevent a click-jacking vulnerability. Click-jacking is a technique, where the attacker uses transparent elements to trick a user into clicking on a button or a link, which is not the mentioned one.

$$v_{36} = \begin{cases} 0 & \text{if X-Frame-Options header is implemented;} \\ 1 & \text{otherwise.} \end{cases}$$

### X-Content-Type

X-Content-Type-Options is HTTP header that prevents content sniffing.

$$v_{37} = \begin{cases} 0 & \text{if X-Content-Type-Options header is implemented;} \\ 1 & \text{otherwise.} \end{cases}$$

### Domain similarity

Attackers may try to impersonate well-known service in the URL. It is because the user may not see the difference between `http://paypal.com` and `http://paypel.com`. For this feature, we will use Google search. We will query our domain into the search engine and if the word is recognized as misspelled, we will compute the string distance with the use of Levenshtein distance [7] between suggested

word and our domain. The final value will be computed with this formula:

$$v_{38} = \begin{cases} 0 & \text{domain is same as suggested word;} \\ \frac{1}{\text{string distance}} & \text{otherwise.} \end{cases}$$

The more similar will be the word (small string distance), then the higher will be the feature value.

#### Autonomous system number

The autonomous system is a collection of IP prefixes under the single administrative entity. We will precompute new property for the ASN based on our datasets. The formula is based on the occurrence of the ASN number in the benign and phishing dataset. Phishing/benign ASN ratio is the count of a specific ASN divided by a total count of all ASNs.

$$v_{39} = \frac{\text{phishing ASN ratio}}{\text{phishing ASN ratio} + \text{benign ASN ratio}}$$

## 3.6 Summary

In this chapter, we have provided a list of features that could be potentially helpful in classifying the site. As we had performed statistics for some of the features, we believe that a lot of them are unnecessary. We will address this problem in the next chapter.

# 4 URL classification

Now that we have some knowledge about phishing detection methods and how prevention works, we can design our system.

We have introduced roughly 50 features, that we can determine from the given URL. However, what to do with this data? How do we know based on these features, whether this is or it is not a phishing URL? For this problem we can look at the machine learning algorithms and try to find a solution there. We are going to find a function that for values $v_1, v_2, ..., v_{39}$ will return a probability that web site is classified as a phishing one.

## 4.1 Machine learning basics

A machine learning algorithm is an algorithm that can learn from data [15]. The common categorization of algorithms is done into several categories: *supervised*, *unsupervised* and *reinforcement learning*.

Unsupervised machine learning algorithms take data as input without any additional previous classification or labeling and they are trying to find some structure or patterns in the data. Typical usage of these algorithms is with neural networks, deep learning, or image labeling.

Supervised machine learning algorithms take not only the data, but also the classification of the data. They are meant to be used on a different set of problems than previous categories, such as categorization or anomaly detection. The algorithms are then fed with the data and classifiers. Since this category of machine learning (ML) algorithms sounds the most promising, we will continue only with this one.

We will use our previously defined features and create for the URL a **feature vector**. We'll run our features on the URL, transform it on the values for the features and add a classification label. We will set that *0* is *benign URL*, *1* is for *phishing URL* (see on fig. 4.1).

We will then prepare training data from the URLs by building a matrix from the feature vectors. We will try to reduce our 49 item

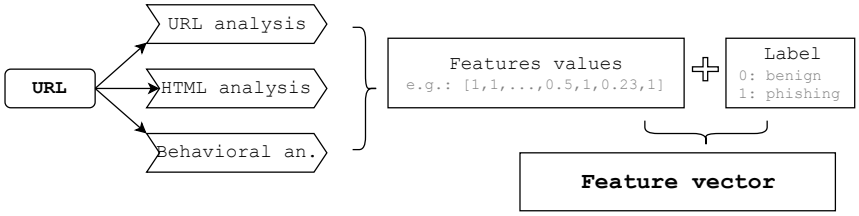


Figure 4.1: Transforming the URL to feature values

large[1] feature set to find out which features are irrelevant to the final result, and then we will train the model. The whole process is on the figure 4.2.

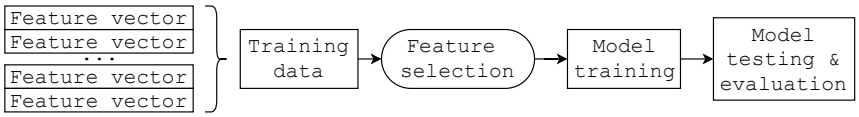


Figure 4.2: Transforming the feature vector into the model

## 4.2 Training data for the machine learning

Since we would like to present the way how we implemented a heuristic model for checking the URL score, we will skip for now the implementation details about transforming URLs into training data. The reader will read more about it in the 5.4.1.

## 4.3 Creating the model

Right now we have thousands of labeled feature vectors consisting of around 50 features, which are called the training data. Since computing of some features may be costly (concerning the delay of networking, API rate limits), we will try to identify irrelevant features.

1. The number is not the 39, since we split several feature value formulas into multiple functions (e.g., we are not mentioning specific feature value for the length of the URL path, but we have implemented it.

For this purpose, we will work with the machine learning toolkit in Python called `scikit-learn` [30]. It is an open-source group of tools for data mining and data analysis provided as an easily accessible library with excellent documentation and examples. For the better user experience and future extendibility, we will use Jupyter Notebooks, which is a web application allowing us saving the report and the code.

### Feature reduction

Feature reduction for the classification problems could be made in several approaches[2]:

- *univariate feature selection* work by selecting the best features based on statistical tests
- *recursive feature elimination* works as recursively trying to test and evaluate the model on a smaller set of the features
- *selecting from model* works as removing features that are considered unimportant based on some threshold

We've selected several classification algorithms for comparison: Logistic Regression, LinearSVC, Perceptron, Lasso, ExtraTreeClassifier, DecisionTreeClassifier and RandomForestClassifier. Other works also used regression algorithms, but we found out that this is wrong since we are trying to classify values and not predict a new value.

During the first iteration for finding relevant features, we had built a list of importance indices for each algorithm for each feature selection approach. Thanks to these results, we have found out that the least 5-10 important features are almost identical for all methods. Also, the 8-10 most essential features are almost identical for all methods. This means that we can find features that we can remove.

We have got a grasp on which features are important and which are not, but we need to be more scientific and evaluate how this feature selection is altering our model. For this action, we need to evaluate our model. In ML there is a technique for this: we will split our training model into two groups – train and test data. It has a drawback that we are reducing our training data by some percentage.

2. `https://scikit-learn.org/stable/modules/feature_selection.html`

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing a CV. In the basic approach, called k-fold CV, the training set is split into k smaller sets. The following procedure is followed for each of the k folds: we run recursive feature elimination with cross-validation and find that for our training-data the best-performing algorithms are RandomForestClassifier and ExtraTreesClassifier with over 94% of accuracy.

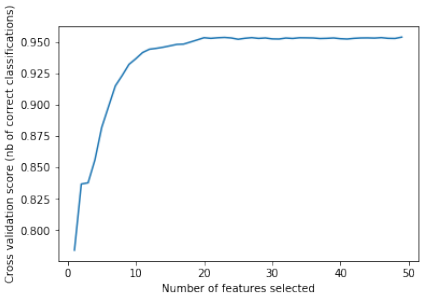


Figure 4.3: Extra tree classifier

The whole process is in the notebook `./data-science/jupyter-notebooks/feature-reduction.ipynb`.

We chose as our final model ExtraTreeClassifier, with the following features: length of the host, special keywords, special length of the path, length of the URL, special characters in the query, special characters in the path, global top level domain, anchor element link, favicon source location, ASN rogue index, domain similarity, DNSSEC, DNS updated and DNS created.

We have serialized our model for later use. It is available in the Python's pickle format at `./data-science/pretrained_model.pkl`.

# 5 Application

We now have all the information needed to create an application, which main focus is returning a phishing score for the given URL.

Let's design an algorithm for this problem (visualization on hte figure 5.1) based on the previous chapters:

1. Look into the database, whether we have already computed phishing score in the past.
2. Check Phishtank. If there is a match, return the phishing score 100.
3. Check Google Safe Browsing. If there is a match, return the phishing score 100.
4. Compute feature vector for relevant features, which we defined in previous chapters:
   (a) Perform URL analysis.
   (b) Perform behavioral analysis.
   (c) Perform HTML analysis.
5. Check prediction model with computed feature vector and return obtained phishing score.
6. Store the result in the database.

## Database

We have chosen PostgreSQL as our database engine. It is a well known, relational, free, and open-source database. It has command-line client `psql` and also bindings to all relevant languages. We will store database server information in the following environmental variables, so we can access the server without any further hassle: `THESIS_DB_HOST`, `THESIS_DB_PORT`, `THESIS_DB_USER`, `THESIS_DB_PASSWORD`, `THESIS_DB_DBNAME`. The database will remain empty as we will write all the setting up commands and table creations in modules.

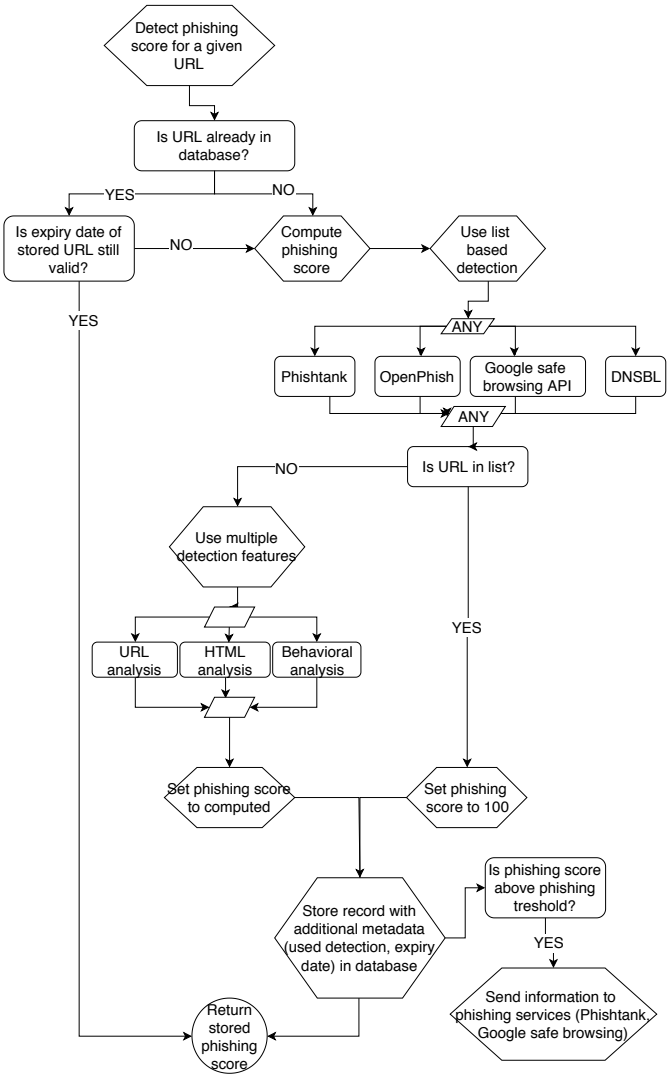


Figure 5.1: Activity diagram

## 5.1 Prediction model module

First, we implement a module, which for given feature values computes a phishing score on the pre-trained model.

### Technical details

Since we have done a data science in Python's ecosystem, we will continue and implement this module in Python.

### Python's virtual environment

For the module implementation the Python 3.7 was used. Python has a tool `pip` for installing different modules and packages in a different version. One of these packages is `virtualenv` which can create a separate environment, so we do not mix up the packages with system-wide packages. The separate environment then can be quickly prepared with previously defined packages in the file `requirements.txt`. This is minimizing the overall time for development on a new machine to a fraction. The full instructions are attached in the `modules/model-checker/README.md`.

### Pretrained model

In the previous chapter, we have concluded that `ExtraTreeClassifier` is the best model for our problem. We have serialized it with Python's module `pickle`, so we do not have to train it whenever we start the application. In our module we will import the binary data `modules/model-checker/pretrained_model.pkl` with the `pickle` library, and then we can use the model as a classic Python's object.

Our model needs these 15 following features: `host_length`, `spec_keywords`, `path_length`, `url_length`, `spec_chars_query`, `spec_chars_path`, `gtld`, `ahref_link`, `favicon_link`, `src_link`, `asn`, `similar_domain`, `dnssec`, `dns_updated`, `dns_created`. The model accepts input as `numpy` array of values in the specific order.

Based on values for these features, the model can predict probabilities for the classes[1]. We are interested in the probability of the

1. Remark: we have defined two classes: benign (0) and phishing (1)

phishing class. We will define the score as the probability in the range [0, 100], where 100 means the phishing site.

### Module interface

Since we will work with different modules and programming languages, we do not want to be locked down in a specific protocol or binary exchange format. We have decided that we will pass the messages through JSON format. It allows us an easy implementation and extensibility in the future. We can also add error messages and propagate it on the client-side, so we will have the correct error message if something unexpected happens.

We have firstly implemented the application with a command-line interface, so that input could be send from a standard input (argument `--stdin`) or from a command-line argument (argument `--data-json <JSON>`). It has the advantage of easy testing and simple usage. However, we have found a big disadvantage – on a basic workstation, importing model and then model prediction takes several seconds, which is for a real-time network application unacceptable.

To solve this problem, we have implemented a simple TCP server using `socketserver` library. We can then define a port (argument `--server <PORT>`), so the module will behave as a service and loads the model just once. In this way, we are eliminating the need for importing the model each time for each request. Example of the (shortened) input and output:

```
>>> input
{
    "ahref_link":1.0,
    "asn":0.0657,
    "url_length":0.027210884353741496,
    ...
}
<<< output
{
    "score": 66
}
```

In case of any errors, we are returning JSON with the following structure:

```
{
```

```
    "error": "MODEL_CHECKER",
    "message": <custom message>
}
```

The last step is packing the application, so we can distribute and use it without setting up a development environment. We are using `pip` package `PyInstaller`, which freezes packages into standalone executable. We wrote shortcuts in the `Makefile`, so packing the application is as simple as:

```
$ make pack
$ bin/model-checker --help
```

## 5.2 HTML analysis module

The second module is for the feature analysis of the web page content.

### Technical details

Because we have implemented this module before we have started working on a previous module written in Python, we have chosen JavaScript for implementation, since it is a fitting language regarding operations with the DOM[2] structure.

### Node.js application

Node.js is a JavaScript runtime environment that runs outside of the browser and it is built on Chrome's v8 JavaScript engine. We will build a command-line application with it. For better portability, we will use a tool `nvm`[3] which allows us an easily installation of a specific version Node.js. We have chosen the latest stable version (v.12.10.0). All the installation steps are available in `modules/html-analysis/README.md`.

The big advantage of the JavaScript ecosystem is its packaging tool `npm`[4]. With its help, we can use third party packages and do not have to implement our application from scratch. Packages required for the execution (with lots of other information, such as shortcut commands

2. Document object model – a representation of the web page as a tree structure
3. Node Version Manager - `https://github.com/nvm-sh/nvm`
4. Node package manager – `https://www.npmjs.com/`

for testing, or metadata) are stored in a file `package.json`. When we want to install the application on any machine, it is sufficient to execute `npm install` and everything should be ready for use.

As we are trying to complement good code quality, we are using the linting tool `eslint`, which has a set of rules that code needs to conform. This way, we can ensure that we have the same coding style through all projects, and if a new person starts working on this project, it will stay concise. We have chosen the `standard`[5] linting package, which used widely for all sorts of projects and has a rising popularity. We can check our source code by executing `npm run lint` to check for problems in our code.

### Module interface

We have built a command-line interface with the use of `yargs`. Initially, we have developed a command-line utility, which would accept the input either from the command line argument (`--url <URL>`) or the standard input (`--stdin`). This version was used at the start when we needed to transform many URLs to the feature values so that we could perform later analysis on our data. The user can specify, which features should be evaluated by command-line flags (e.g. `--feat-input-tag`, `--feat-popup-window`). The complete user manual and available commands are in the help manual (`--help`). The first output that we used was in the `csv` format, so we could easily import it into the Python for data science.

This solution has been sufficient for performing data science at the start, but it has a drawback of starting up the Node.js engine in several seconds, which is unacceptable in real-time applications. We have then also incorporated a simple TCP server that would listen on a socket for the JSONs (argument `--server <PORT>`). The JSON would contain the same information as would the command line, here is the example of the input:

```
>>> input
{
    "url": "http://google.com",
    "features": {
        "featSrcLink": true,
```

5. `https://standardjs.com/`

```
        "featAhrefLink": true
    }
}
<<< output
{
    "src_link":0,
    "ahref_link":0.7058823529411765
}
```

If any error occurs, we will reply with the JSON containing `"error"` field so that the client application can handle it.

#### Feature implementation

For the implementation of the features, we need to operate with a web page. Firstly, we use a `node-fetch` package to download a page, and then we will parse it and create a DOM with the use of `jsdom`. Then we have access to the DOM as we would have in the browser and we can use all of the `document` manipulation functions[6]. The page may be down, or we do not have access to it – in this case, we are setting all features to -1, so the next client knows that the results are unusable.

We had implemented all the features in the section 3.3 in this module, the source code is in the `modules/html-analysis/src/page.js`. There were five features (input tag, form handler, invisible iframe, popup window, miss-leading link) that we need to evaluate before we would know how to compute our feature value. We have provided an argument `--include-values` so we were able to perform data analysis.

We have written unit tests in the `mocha`[7] for all of the features, so we are sure that they are computing the values that we are expecting. Tests could be executed with the command `npm run test`.

## 5.3 Phishtank updater module

In section 2.3, we have mentioned that we were gathering the Phishtank data with a script. We had made a JavaScript utility, which would

6. `https://developer.mozilla.org/en-US/docs/Web/API/Document`
7. `https://mochajs.org/`

process this data and based on provided metadata detect, when the web site went down.

The next feature of this utility is that it download the current snapshot of the Phishtank database, compare it with our database and update it[8] accordingly. We have chosen the package `typeorm`[9] for the communication with the database.

The object-relation mapping (ORM) is for a convenience usage of the database, so we do not need to write raw queries. Instead of that, we can work with database records as with objects. The `typeorm` needs a configuration[10] to work properly, mainly database connection information (the database host, port, user, password and database name). We will store all configuration values in the project-wide `.env` file, which will pass environmental values to the application, and in case of customization, we can override it from the environment by exporting specific one (e.g., `export DB_USER=custom user;`). This method is used for using different configurations between the deployment and the development phase.

We have defined two tables in the database, `phishtank` and `last_updated`. We have created for both of the a class representing the table and wrote the migration script so that we can create a schema for the table in an automated way during the initial setup. We can also explicitly run migrations from the command-line with our shortcut command `npm run migrate`.

We will run this script in the update mode (`node src/index.js --update`) periodically with the `cron`[11], so we will have a fresh data every day. The user also needs to define the `PHISHTANK_API_KEY` variable in the environment.

## 5.4 Phishing score module

We have now all tidbits that we need to create the final application, which we have designed in figure 5.1.

---

8. Source code: `modules/update-phishtank/src/operations/phishtank.js`
9. `https://github.com/typeorm/typeorm`
10. `https://github.com/typeorm/typeorm/blob/0.2.21/docs/using-ormconfig.md`
11. Task scheduler for Unix-like systems

### C++ ecosystem

The partner of this work is developing network applications written in C/C++ language; therefore, we have chosen as the primary language C++, so the possibility of the future integration of this application can be performed.

Compared to previously used languages (JavaScript, Python), C++ is not an interpreted but a compiled language. That means that the source code that we write will be transformed into the machine code. The whole process is divided into three steps: preprocessing of the source code, a compilation of the source code into object files, and then linking. This whole process is called building.

For small projects (having low number of dependencies and classes), it is usually sufficient to build an application just with a compiler. However, a more robust project that needs to be maintained and easily extensible, needs additional tools.

In many languages, the topic about the ecosystem and easy development is solved since they have proper tooling (Python has `pip`, JavaScript has `npm/yarn`). Unfortunately, this is not the case in the C++ world.

Since we are targeting Ubuntu as our main platform, we have chosen `g++` as our compiler. Regarding the build automation tools, the standard for many years on the Linux platform was using a `make`, where we would define targets with exact steps for execution.

As the project grows, it is hard for a maintenance. Because of that, higher-level build system has been developed. Examples of these tools are `autotools`, `qmake`, `CMake`, `meson`. These tools are generating `Makefiles` for the `make` build system, but they have some additional features such as targeting a compiler, file search, custom functions, and many others. We have chosen for this project `CMake` because it is becoming a sort of standard in the C++ ecosystem.

The CMake project is using `CMakeLists.txt`[12] where we define how is the final executable built. Writing a readable and maintainable `CMake` project has showed up as tedious task – if a reader is interested, we are referring to a Professional CMake [57]. Steps for building a `CMake` application are following:

12. Source code: `modules/phishscore/CMakeLists.txt`

```
$ mkdir build && cd build
$ cmake ..
$ make
```

We do not want to write URL parsing, database operations, and many others functionalities from the scratch, as we would make many bugs, and it would be out of the scope for this work. We were looking for a way, how to integrate third party libraries into our project and found several options how to handle dependencies: `git submodules`, `vcpkg` (written in C++/CMake) and `conan` (written in Python). We have chosen the `vcpkg` since all libraries that we knew we will need are available there.

The `vcpkg` is multiplatform library manager from the Microsoft. After installing and integrating the `vcpkg` into the system, we can install libraries with following command: `vcpkg install <PKG NAME>`. After installing the package we will see a message with following instructions, how to incorporate a library into our `CMakeLists.txt`:

```
    $ vcpkg install gtest

The package gtest is compatible with built-in CMake targets:
    enable_testing()

    find_package(GTest CONFIG REQUIRED)
    target_link_libraries(main PRIVATE GTest::gtest GTest::gtest_main
                                       GTest::gmock GTest::gmock_main)
```

After the installation of the `vcpkg`, we need to set the environmental variable `CMAKE_VCPKG_TOOLCHAIN` so the CMake project can find installed libraries.

Without this tool, we would have to manually deep into the documentation of each library, find steps for building and integrating. This is a common problem that adds up unnecessary overhead for new developers in C++.

Combining all these prerequisites can take a lot of time for the new project, but once it is prepared, the future development, and the maintenance is quite easy, and can be easily integrated into other projects and systems.

### URL library

Our application needs to work with a URL. Unfortunately, the C++ standard library to this date still does not have a library, which we could use for parsing URLs out of the box. The initial idea was to

implement parsing a URL by ourself, but this task was too complex so we decided to switch for third party library.

We have made a list of known libraries and measured the time and the memory footprint for each of them while parsing 1 million URLs. Results are in table 5.1.

Table 5.1: Comparison of URL libraries in C++

| Library name | Parsed | Elapsed time | Memory (relative) |
|---|---|---|---|
| C++ Rest SDK [61] | 98.83% | 5.45s | 1.46 |
| Folly [13] | **100.00%** | 8.14s | 1.63 |
| C++ Network URI [10] | 98.41% | 24.70s | 1.35 |
| Poco [62] | 99.98% | 6.89s | 1.27 |
| Skyr URL [60] | 99.99% | 352.11s | 2.24 |
| uriparser [64] | 98.41% | **1.99s** | **1** |

Size of the input set: 1,000,000 URLs

Based on factors such as the percentage of correctly parsed URLs without throwing an exception, the time elapsed, the memory footprint, the library interface, the documentation, and GitHub stars, we have chosen the library **Poco**[13]. The experiment is available in `./benchmark-urls`.

### 5.4.1 Stage one: the creation of the training data

For building a command-line interface in the C++ application, we have chosen the library `cxxopts`. The command-line interface is implemented in the class `cmdline.h`. We had several working versions of passing options and we found out that environmental variables combined with command line arguments suites this application the best. After parsing the command line, we create a `options` object and pass the reference to our classes.

We had implemented utility for transformation a database table containing only URLs to the new table with parsed URLs parts. We

13. `https://pocoproject.org/`

have done this to be able to do an analysis of our data in chapter 3. For this part, we are using libraries `libpqxx` and `Poco::URI`.

After that, we moved to the implementation for the transformation a URL to the feature vector, like we showed in the figure 4.1. What is the process to get the feature vector? The high-level view is following: Input URLs → parse it to the `Poco::URI` object → create `training_data` object → create `feature_vector` objects for each URL → feature vector will create `url_features`, `html_features` and `host_based_features`.

### Classes `training_data` and `feature_vector`

The training data class represents the transformation of URLs to the `csv` format needed by the Python module `model-checker`. We can define several options, such as how we are taking input (either URLs from the command-line (`--td-url`) or standard input (`--stdin`) or what features do we want to test (all feature options are in help section "Feature options"). We also need to define a path to the HTML analysis module from the previous section by the argument `--html-analysis-path`. An example of usage execution for the full command is on page 69 in the appendix. We are also providing `make` target, so during the testing or development, a user does not have to write all arguments over and over.

The feature vector class is combining all methods from the chapter 3. For each category from the detection methods, we will create a new class. We would like to separate them, since they have logically different meaning. The user can then define what features needs to be checked. Based on these arguments (e.g. `--feat-asn --feat-ip-address`) we will build our internal represenation with flags symbolizing, what features are used. The definition of all features with its column mapping for the `csv` file is available in the `modules/phishscore/src/features/feature_enum.h`. The two previous features would be represented as follows:

```
1  uint64_t flags = feature_enum::asn | feature_enum::ip_address; // bitwise OR
```

This was designed with the idea of easy implementation for new features in the future.

### Class `url_features`

This class is implementation of section 3.2. Thanks to the previous data analysis, this class is straightforward implementation using mainly regular expressions and string methods.

### Class `html_features`

This class is a wrapper for the HTML analysis module from the previous section. We need to pass either path to the binary (`--html-analysis-path <PATH>`), or port (`--html-analysis-port <PORT>`) if we are running it as a service. After obtaining the input, we will process it and adds up to the feature vector.

### Class `host_based_features`

This is the last part of the features. It's implementing the section 3.5 using following programs: `dig`[14], `whois`[15], `openssl`[16], `curl`[17].

All these tools are using network queries, and the overall user-experience of our final application would be degraded if we need to wait synchronously for finishing all network calls. Because of that, during the initialization of the class, we are spawning a new thread for each of the network calls and storing it for the given URL. Here is the example snippet, how are we handling the situation, that the feature ASN needs a resolved IP address beforehand:

```
std::vector<std::thread> threads;
std::thread dig_thread;
if (_flags & (feature_enum::dns_a_record | feature_enum::dnssec | feature_enum::asn)) {
    dig_thread = std::thread(&host_based_features_t::fill_dig_response, this);
}
if (dig_thread.joinable()) {
    dig_thread.join();
}
if (_flags & (feature_enum::asn)) {
    // we need to have dig response before we can operate with SLD
    // because we need to have resolved IP address
    threads.push_back(std::thread(&host_based_features_t::fill_asn, this));
}
```

14. DNS lookup utility
15. Client for the whois service
16. Client for checking HTTPS certificate
17. Client for obtaining HTTP response headers

### Unit testing

We have written unit tests for all features. This ensures that we are computing the value that we are expecting. We are using `gtest`[18] as the testing framework. If we want to build tests, we need to pass a variable to the CMake. After they are built, the execution is as following:

```
$ mkdir build && cd build
$ cmake -DRUN_TESTS:BOOL=ON ..
$ make
$ tests/all_tests  # Binary containing all tests
```

## 5.4.2 Stage two: phishing score

In section 5.1, we have written the list of features that we will use in the application. We will use 15 features out of 49 possible. We can now flatten our runtime application.

### Class `model_checker`

This class is a wrapper for the model checking module in section 5.1. We need to set either path to the module (`--model-checker-path`) or port in the case, when we run the model as a service (`--model-checker-port`).

An example of the usage is on page 69 in the appendix.

### Class `safebrowsing_api`

For the usage of Google's API, one needs to register on the Google API console website[19]. After obtaining the API key, our application will load it from the environmental variable `GOOGLE_SAFEBROWSING_API_KEY`.

This class is wrapper for the provided lookup API[20]. The user code with this class is simple as:

```
    std::string sb_api_key = get_env_var("GOOGLE_SAFEBROWSING_API_KEY");
    safebrowsing_api sb_api(sb_api_key);
    bool is_unsafe = sb_api.check_unsafe_url(opts.input.url);
```

18. Google Test Framework
19. `https://console.developers.google.com/`
20. `https://developers.google.com/safe-browsing/v4/lookup-api`

### Module interface

We put everything together and simplified it to the command line argument `--check-url`, which will run the algorithm from the start of the chapter (figure 5.1). For the simplicity, we can define most values in the environment, so we do not have to write all over the command line arguments. The example is provided on the page 69 in the appendix.

We have also added implemented a simple TCP server and possibility to run as service (`--server <PORT>`). We are waiting for incoming JSON requests in format `{"url": <URL>}` and then returning `{"url": <URL>, "score": <SCORE>}`.

The full manual page with all possible arguments are available in `--help`. The examples how to run an application could be found in `Makefile`.

# 6 Deployment

We were trying to prepare an application that is easily maintainable and deployable with minimal effort. For this task, we have chosen the Docker as our packaging tool.

### Docker

Docker is an open-source project for easy creating, running, and deploying containers. Containers allow a developer to package an application, with all dependencies and resources such as libraries, and ship it as one package. It is similar to a virtual machine, but the container is sharing a Linux kernel with a host, so it does not consume as much resources as a virtual machine.

The usual workflow during the work with containers is that we will write a Dockerfile, then we build a Docker image from that file. When we have a Docker image, we can create instances – containers. Here is a simple example:

```
$ cat Dockerfile
FROM ubuntu
CMD ["sh", "-c", "echo Hello from Docker"]
$ # Build Docker image in current directory with name hello-world
$ docker build -f Dockerfile -t image/hello-world .
$ # Run Docker container from built image
$ docker run --rm image/hello-world
Hello from Docker
```

We have written a Dockerfile for each module in our repository, and we are using a `postgres:11.2` image as our database. We have together five services: the `model-checker`, the `html-analysis`, the `phishtank-updater`, the `phishscore` and the database. We have incorporated two techniques to have as smallest container as possible: using a `.dockerignore` file and using a multi-stage builds.

When Docker is building an image, it sends by default whole directory to the context of the Docker engine. We can alter this behavior when we want to ignore build artifacts from the host system (such as `node_modules`, `venv`).

The multi-stage build is for separating environments for a service. Usually, we need to have a building environment where we have all the dependencies and libraries, and then the runtime environment, where we have only compiled binary and its linked libraries, which it

needs for execution. Thanks to this, we can lower resulting image size for the `phishscore` image from 2.9 GB to 400 MB[1]

### Docker Compose

The `docker-compose`[2] is tool for organizing containers in a multi-container environment. The user writes a config file (`docker-compose.yml`) with definitions for services. Here is an example of the same functionality from before:

```
$ cat docker-compose.yml
version: '3.7'
services:
    hello-world:
        image: image/hello-world
        build:
            context: .
$ docker-compose build
$ docker-compose up
Hello from Docker
```

We will leverage the `docker-compose` feature, that it will create a private network for all defined services in one configuration file. We can then easily communicate between our services. The customization for arguments is done through an `.env` file. If we want to hide some parameters from the public domain (e.g., API keys), we can override it from the environment.

### Running the final application

We have prepared the `.env` file in the root directory. Several variables are intentionally left blank as template, since they are system-specific or should remain secret:

```
# CMAKE_VCPKG_TOOLCHAIN=      # Path where is stored vcpkg cmake integration file
# THESIS_MODEL_CHECKER_PROG= # Path to the model checker binary
# THESIS_HTML_ANALYSIS_PROG= # Path to the HTML analysis binary -
# THESIS_DB_DOCKER_VOLUME=    # Path to the folder where postgres will persist data
# PHISHTANK_API_KEY=
# GOOGLE_SAFEBROWSING_API_KEY=
```

Containers are typically self-contained without side effects. For the full operability of our application, we need to create a new empty directory and set it to `THESIS_DB_DOCKER_VOLUME`. `docker-compose` then creates a `docker volume` with this path, which is used for the

1. This could be optimized more if needed (e.g., using Release build, using a smaller base image).
2. `https://docs.docker.com/compose/`

Postgres database. Then, after stopping the container or rebooting the system, we would not lose the data.

Variables `PHISHTANK_API_KEY` and `GOOGLE_SAFEBROWSING_API_KEY` will be available in the attachments so that the reader can test the whole application without registration. We need to export it into the environment with the (`export PHISHTANK_API_KEY=<KEY>`) command.

After preparing these two steps, we can build the application:

```
$ docker-compose up
```

The first run will take a long time: it needs to download bare images, and then it takes much time for building all dependencies and libraries. If we want to save time and skip the building phase, we can download pre-build images which we had pushed to the Docker hub before:

```
$ docker-compose pull
$ docker-compose up
```

We can test our application with a simple telnet client:

```
$ telnet localhost ${THESIS_PHISHSCORE_PORT}
>>>>>
{
    "url": "http://google.com"
}
<<<<<
{
    "url": "http://google.com",
    "score": 22
}
```

# 7 Summary

The main task of this thesis was to design and implement a system, which will decide whether the given URL is considered as phishing or not.

We had introduced the phishing problem in the second chapter. We have looked at phishing trends and possible defense mechanisms for a user.

In the next chapter, we have analyzed academic papers and expert researches to find out how we can design an anti-phishing system. We have compared different approaches and listed their pros and cons.

In the fourth chapter, we had developed a heuristic model for the URL classification. We have created data sets based on the features from the third chapter, tested and compared different machine learning classification algorithms. Based on the results, we chose the best model for later use.

In the fifth chapter, we had designed our anti-phishing system. We have splitted the system into smaller modules which can cooperates together. We have provided two main functionalities: creating the training data for machine learning algorithms and then evaluating a phishing score for the URL.

In the last chapter, we have described the way how we can distribute the application and how we can run it on the network. We have implemented it in the way that there is almost zero configuration. It is easy to run in the existing network.

The output of this work is the anti-phishing system, which for the given URL is returning probability whether the URL is phishing one. The system can be easily deployed on the network and can be extended with new features. The process of adding new features into the system is: implement a new feature, recreate training data, train, and evaluate the model. All these steps are easily reproducible in our system.

The client applications on the network can implement a TCP communication with our system. After receiving the phishing score from our system, they can decide their follow-up action whether they would allow or block the communication.

# Bibliography


[1] *2019 Data Breach Investigations Report*. Verizon. 2019. URL: https://enterprise.verizon.com/resources/reports/2019-data-breach-investigations-report.pdf (visited on 11/14/2019).

[2] Sahar Abdelnabi, Katharina Krombholz, and Mario Fritz. "WhiteNet: Phishing Website Detection by Visual Whitelists". In: *arXiv e-prints*, arXiv:1909.00300 (Aug. 2019), arXiv:1909.00300. arXiv: 1909.00300 [cs.CR].

[3] Abdullah Alnajim and Malcolm Munro. "An Anti-Phishing Approach that Uses Training Intervention for Phishing Websites Detection". In: *2009 Sixth International Conference on Information Technology: New Generations* (2009). DOI: 10.1109/itng.2009.109.

[4] *APWG. Unifying The Global Response To Cybercrime*. URL: https://apwg.org/ (visited on 11/14/2019).

[5] R. Aravindhan et al. "Certain investigation on web application security: Phishing detection and phishing target discovery". In: *2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS)*. Vol. 01. Jan. 2016, pp. 1–10. DOI: 10.1109/ICACCS.2016.7586405.

[6] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. http://www.rfc-editor.org/rfc/rfc3986.txt. RFC Editor, Jan. 2005. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[7] Paul E. Black. *Dictionary of Algorithms and Data Structures. Algorithms and Theory of Computation Handbook: Levenshtein distance*. CRC Press LLC, 1999. URL: https://www.nist.gov/dads/HTML/Levenshtein.html.

[8] G. Chakraborty and T. T. Lin. "A URL address aware classification of malicious websites for online security during websurfing". In: *2017 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. Dec. 2017, pp. 1–6. DOI: 10.1109/ANTS.2017.8384155.

[9] T. Churi et al. "A secured methodology for anti-phishing". In: *2017 International Conference on Innovations in Information, Em-*

*bedded and Communication Systems (ICIIECS)*. Mar. 2017, pp. 1–4. DOI: 10.1109/ICIIECS.2017.8276081.

[10] *cpp-netlib*. network-uri. 2013. URL: https://github.com/cpp-netlib/uri (visited on 11/28/2019).

[11] *CyberArk Global Advanced Threat Landscape Report 2018. The Cyber Security Inertia Putting Organizations at Risk | CyberArk*. URL: https://www.cyberark.com/resource/cyberark-global-advanced-threat-landscape-report-2018/ (visited on 11/14/2019).

[12] *ENTERPRISE PHISHING RESILIENCY and DEFENSE REPORT. 2017*. PhishMe. URL: https://cofense.com/wp-content/uploads/2017/11/Enterprise-Phishing-Resiliency-and-Defense-Report-2017.pdf (visited on 11/14/2019).

[13] *Folly*. Facebook. 2013. URL: https://github.com/facebook/folly (visited on 11/28/2012).

[14] Guang-Gang Geng et al. "Favicon - a clue to phishing sites detection". In: *2013 APWG eCrime Researchers Summit*. Sept. 2013, pp. 1–10. DOI: 10.1109/eCRS.2013.6805775.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[16] *Google Safe Browsing*. URL: https://developers.google.com/safe-browsing/v4/ (visited on 05/07/2019).

[17] Christopher Hadnagy. *Social engineering: the science of human hacking*. Wiley, 2018.

[18] V. R. Hawanna, V. Y. Kulkarni, and R. A. Rane. "A novel algorithm to detect phishing URLs". In: *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT)*. Sept. 2016, pp. 548–552. DOI: 10.1109/ICACDOT.2016.7877645.

[19] M. Khonji, Y. Iraqi, and A. Jones. "Phishing Detection: A Literature Survey". In: *IEEE Communications Surveys Tutorials* 15.4 (Sept. 2013), pp. 2091–2121. ISSN: 1553-877X. DOI: 10.1109/SURV.2013.032213.00009.

[20] R. Kumar et al. "Malicious URL detection using multi-layer filtering model". In: *2017 14th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*. Dec. 2017, pp. 97–100. DOI: 10.1109/ICCWAMTIP.2017.8301457.

[21] Justin Ma et al. "Beyond blacklists: learning to detect malicious Web sites from suspicious URLs". In: Jan. 2009, pp. 1245–1254. DOI: 10.1145/1557019.1557153.

[22] S. Marchal et al. "Off-the-Hook: An Efficient and Usable Client-Side Phishing Prevention Application". In: *IEEE Transactions on Computers* 66.10 (Oct. 2017), pp. 1717–1733. DOI: 10.1109/TC.2017.2703808.

[23] S. Marchal et al. "PhishStorm: Detecting Phishing With Streaming Analytics". In: *IEEE Transactions on Network and Service Management* 11.4 (Dec. 2014), pp. 458–471. ISSN: 1932-4537. DOI: 10.1109/TNSM.2014.2377295.

[24] P. Mockapetris. *Domain names - implementation and specification*. STD 13. http://www.rfc-editor.org/rfc/rfc1035.txt. RFC Editor, Nov. 1987. URL: http://www.rfc-editor.org/rfc/rfc1035.txt.

[25] R. M. Mohammad, F. Thabtah, and L. McCluskey. "An assessment of features related to phishing websites using an automated technique". In: *2012 International Conference for Internet Technology and Secured Transactions*. Dec. 2012, pp. 492–497.

[26] *OpenPhish*. URL: https://openphish.com/index.html (visited on 09/07/2019).

[27] Y. Pan and X. Ding. "Anomaly Based Web Phishing Page Detection". In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. Dec. 2006, pp. 381–392. DOI: 10.1109/ACSAC.2006.13.

[28] S. Parekh et al. "A New Method for Detection of Phishing Websites: URL Detection". In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. Apr. 2018, pp. 949–952. DOI: 10.1109/ICICCT.2018.8473085.

[29] S. Patil and S. Dhage. "A Methodical Overview on Phishing Detection along with an Organized Way to Construct an Anti-Phishing Framework". In: *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*. Mar. 2019, pp. 588–593. DOI: 10.1109/ICACCS.2019.8728356.

[30] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[31] *Phishing Activity Trends Report. 4th Quarter 2012*. Anti-Phishing Working Group. 2013. URL: https://docs.apwg.org/reports/apwg_trends_report_Q4_2012.pdf (visited on 09/26/2019).

[32] *Phishing Activity Trends Report. 1st Quarter 2013*. Anti-Phishing Working Group. 2013. URL: https://docs.apwg.org/reports/apwg_trends_report_q1_2013.pdf (visited on 09/26/2019).

[33] *Phishing Activity Trends Report. 2nd Quarter 2013*. Anti-Phishing Working Group. 2013. URL: https://docs.apwg.org/reports/apwg_trends_report_q2_2013.pdf (visited on 09/26/2019).

[34] *Phishing Activity Trends Report. 3rd Quarter 2013*. Anti-Phishing Working Group. 2014. URL: https://docs.apwg.org/reports/apwg_trends_report_q3_2013.pdf (visited on 09/26/2019).

[35] *Phishing Activity Trends Report. 4th Quarter 2013*. Anti-Phishing Working Group. 2014. URL: https://docs.apwg.org/reports/apwg_trends_report_q4_2013.pdf (visited on 09/26/2019).

[36] *Phishing Activity Trends Report. 1st Quarter 2014*. Anti-Phishing Working Group. 2014. URL: https://docs.apwg.org/reports/apwg_trends_report_q1_2014.pdf (visited on 09/26/2019).

[37] *Phishing Activity Trends Report. 2nd Quarter 2014*. Anti-Phishing Working Group. 2014. URL: https://docs.apwg.org/reports/apwg_trends_report_q2_2014.pdf (visited on 09/26/2019).

[38] *Phishing Activity Trends Report. 3rd Quarter 2014*. Anti-Phishing Working Group. 2015. URL: https://docs.apwg.org/reports/apwg_trends_report_q3_2014.pdf (visited on 09/26/2019).

[39] *Phishing Activity Trends Report. 4th Quarter 2014*. Anti-Phishing Working Group. 2015. URL: https://docs.apwg.org/reports/apwg_trends_report_q4_2014.pdf (visited on 09/26/2019).

[40] *Phishing Activity Trends Report. 1st-3rd Quarters*. Anti-Phishing Working Group. 2015. URL: https://docs.apwg.org/reports/apwg_trends_report_q1-q3_2015.pdf (visited on 09/26/2019).

[41] *Phishing Activity Trends Report. 4th Quarter 2015*. Anti-Phishing Working Group. 2016. URL: https://docs.apwg.org/reports/apwg_trends_report_q4_2015.pdf (visited on 09/26/2019).

[42] *Phishing Activity Trends Report. 1st Quarter 2016*. Anti-Phishing Working Group. 2016. URL: https://docs.apwg.org/reports/apwg_trends_report_q1_2016.pdf (visited on 09/26/2019).

[43] *Phishing Activity Trends Report. 2nd Quarter 2016*. Anti-Phishing Working Group. 2016. URL: https://docs.apwg.org/reports/apwg_trends_report_q2_2016.pdf (visited on 09/26/2019).

[44] *Phishing Activity Trends Report. 3rd Quarter 2016*. Anti-Phishing Working Group. 2017. URL: https://docs.apwg.org/reports/apwg_trends_report_q3_2016.pdf (visited on 09/26/2019).

[45] *Phishing Activity Trends Report. 4th Quarter 2016*. Anti-Phishing Working Group. 2017. URL: https://docs.apwg.org/reports/apwg_trends_report_q4_2016.pdf (visited on 09/26/2019).

[46] *Phishing Activity Trends Report. 1st Half 2017*. Anti-Phishing Working Group. 2017. URL: https://docs.apwg.org/reports/apwg_trends_report_h1_2017.pdf (visited on 09/26/2019).

[47] *Phishing Activity Trends Report. 3rd Quarter 2017*. Anti-Phishing Working Group. 2018. URL: https://docs.apwg.org/reports/apwg_trends_report_q3_2017.pdf (visited on 09/26/2019).

[48] *Phishing Activity Trends Report. 4th Quarter 2017*. Anti-Phishing Working Group. 2018. URL: https://docs.apwg.org/reports/apwg_trends_report_q4_2017.pdf (visited on 09/26/2019).

[49] *Phishing Activity Trends Report. 1st Quarter 2018*. Anti-Phishing Working Group. 2018. URL: https://docs.apwg.org/reports/apwg_trends_report_q1_2018.pdf (visited on 09/26/2019).

[50] *Phishing Activity Trends Report. 2nd Quarter 2018*. Anti-Phishing Working Group. 2018. URL: https://docs.apwg.org/reports/apwg_trends_report_q2_2018.pdf (visited on 09/26/2019).

[51] *Phishing Activity Trends Report. 3rd Quarter 2018*. Anti-Phishing Working Group. 2019. URL: https://docs.apwg.org/reports/apwg_trends_report_q3_2018.pdf (visited on 09/26/2019).

[52] *Phishing Activity Trends Report. 4th Quarter 2018*. Anti-Phishing Working Group. 2019. URL: https://docs.apwg.org/reports/apwg_trends_report_q4_2018.pdf (visited on 09/26/2019).

[53] *Phishing Activity Trends Report. 1st Quarter 2019*. Anti-Phishing Working Group. 2019. URL: https://docs.apwg.org/reports/apwg_trends_report_q1_2019.pdf (visited on 09/26/2019).

[54] *Phishing Activity Trends Report. 2nd Quarter 2019*. Anti-Phishing Working Group. 2019. URL: https://docs.apwg.org/reports/apwg_trends_report_q2_2019.pdf (visited on 09/26/2019).

[55] *PhishTank*. URL: https://www.phishtank.com/ (visited on 05/18/2019).

[56] N. Sanglerdsinlapachai and A. Rungsawang. "Using Domain Top-page Similarity Feature in Machine Learning-Based Web Phishing Detection". In: *2010 Third International Conference on Knowledge Discovery and Data Mining*. Jan. 2010, pp. 187–190. DOI: 10.1109/WKDD.2010.108.

[57] Craig Scott. *Professional CMake: A Practical Guide*. 1st Edition. Crascit, 2018.

[58] Steve Sheng et al. "An Empirical Analysis of Phishing Blacklists". In: (Jan. 2009).

[59] H. Shirazi, K. Haefner, and I. Ray. "Fresh-Phish: A Framework for Auto-Detection of Phishing Websites". In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. Aug. 2017, pp. 137–143. DOI: 10.1109/IRI.2017.40.

[60] *Skyr URL*. cpp-netlib. 2018. URL: https://github.com/cpp-netlib/url (visited on 11/28/2019).

[61] *The C++ REST SDK*. Microsoft. 2013. URL: https://github.com/Microsoft/cpprestsdk (visited on 11/28/2019).

[62] *The POCO C++ libraries*. POCO. 2012. URL: https://github.com/pocoproject/poco (visited on 11/28/2019).

[63] *University of New Brunswick. URL 2016 | Datasets | Research | Canadian Institute for Cybersecurity | UNB*. URL: https://www.unb.ca/cic/datasets/url-2016.html.

[64] *uriparser*. uriparser. 2018. URL: https://github.com/uriparser/uriparser (visited on 11/28/2019).

[65] Wikipedia contributors. *DNSBL — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=DNSBL&oldid=906187839. [Online; accessed 7-September-2019]. 2019.

[66] Wikipedia contributors. *DNSWL — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=DNSWL&oldid=860892818. [Online; accessed 7-September-2019]. 2018.

[67] Guang Xiang et al. "CANTINA+: A Feature-Rich Machine Learning Framework for Detecting Phishing Web Sites". In: *ACM Trans. Inf. Syst. Secur.* 14 (Sept. 2011), p. 21. DOI: 10.1145/2019599.2019606.

# A Appendix

## A.1 Training data

```
$ phishsvc \
        --enable-training-data \
        --td-url "http://google.com" \
        --td-class-value 0 \
        --enable-features \
        --td-output-url \
        --htmlfeatures-bin /home/astaruch/Documents/code/master-thesis/modules/html-analysis/bin/html-analysis \
          --feat-ip-address --feat-url-length --feat-host-length --feat-path-length --feat-query-length --feat-fragment-length --feat-user-info --feat-domain-count --feat-https-used --feat-extra-https --feat-shortening-service --feat-non-std-port --feat-spec-chars-path --feat-spec-chars-query --feat-spec-chars-fragment --feat-spec-chars-host --feat-gtld --feat-www-prefix --feat-four-numbers --feat-spec-keywords --feat-punycode \
          --feat-input-tag --feat-src-link --feat-form-handler --feat-invisible-iframe --feat-rewrite-statusbar --feat-disable-rightclick --feat-ahref-link --feat-popup-window --feat-favicon-link --feat-old-technologies --feat-missleading-link --feat-hostname-title \
          --feat-redirect --feat-google-index --feat-dns-a-record --feat-dnssec --feat-dns-created --feat-dns-updated --feat-ssl-created --feat-ssl-expire --feat-ssl-subject --feat-hsts --feat-xss-protection --feat-csp --feat-x-frame --feat-x-content-type --feat-asn --feat-similar-domain

url,ip_address,url_length,host_length,path_length,query_length,fragment_length,user_info,domain_count,https_used,extra_https,shortening_service,non_std_port,spec_chars_path,spec_chars_query,spec_chars_fragment,spec_chars_host,gtld,www_prefix,four_numbers,spec_keywords,punycode,input_tag,src_link,form_handler,invisible_iframe,rewrite_statusbar,disable_rightclick,ahref_link,popup_window,favicon_link,old_technologies,missleading_link,hostname_title,redirect,google_index,dns_a_record,dnssec,dns_created,dns_updated,ssl_created,ssl_expire,ssl_subject,hsts,xss_protection,csp,x_frame,x_content_type,asn,similar_domain,label

"http://google.com",0.0,0.0204082,0.0,0.0,0.0,0.0,0.0,0.0,
1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.2,0.0,
0.0,0.0,0.0,0.0,0.0,0.705882,0.0,0.0,1.0,0.05,0.0,1.0,0.0,0.0,
1.0,0.0,0.987518,1.0,1.0,1.0,1.0,1.0,1.0,0.0,1.0,0.279363,0.0,0.0
```

## A.2 Model checking

```
$ phishsvc \
        --enable-model-checking \
        --input-stdin \
        --mc-htmlfeatures-bin /home/astaruch/Documents/code/master-thesis/modules/html-analysis/bin/html-analysis \
        --mc-model-checker-path /home/astaruch/Documents/code/master-thesis/modules/model-checker/bin/model-checker \
Executing command: /home/astaruch/Documents/code/master-thesis/modules/model-checker/bin/model-checker --data-json "{\"ahref_link\":0.08196721311475409,\"asn\":0.548231233822261,\"dns_created\":0.3122684201931035,\"dns_updated\":0.9601617368271942,\"dnssec\":0.0,\"favicon_link\":0.0,\"gtld\":0.0,\"host_length\":0.1111111111111111,\"path_length\":0.0,\"similar_domain\":0.3333333333333333,\"spec_chars_path\":0.0,\"spec_chars_query\":0.0,\"spec_keywords\":0.0,\"src_link\":0.0,\"url_length\":0.04081632653061224}"
71
```

## A.3 Phishscore module execution

```
$ phishsvc --input-url 'http://google.com' --mc-model-checker-path $THESIS_MODEL_CHECKER_PROG --mc-htmlfeatures-bin $THESIS_HTML_ANALYSIS_PROG

[2019-12-09 12:24:18.545] [info] Starting application to check 'http://google.com'
[2019-12-09 12:24:18.546] [info] Retrieving connection string to a database
[2019-12-09 12:24:18.546] [info] THESIS_DB_CONN_STRING=
[2019-12-09 12:24:18.546] [info] THESIS_DB_HOST=0.0.0.0
[2019-12-09 12:24:18.546] [info] THESIS_DB_PORT=11000
```

```
[2019-12-09 12:24:18.546] [info] THESIS_DB_DBNAME=phishing-app
[2019-12-09 12:24:18.546] [info] THESIS_DB_USER=thesis-user
[2019-12-09 12:24:18.546] [info] THESIS_DB_PASSWORD=thesis-password
[2019-12-09 12:24:18.546] [info] THESIS_DB_TIMEOUT=2
[2019-12-09 12:24:18.546] [info] CONN_STRING=host = '0.0.0.0' port = '11000' dbname = 'phishing-app' user = 'thesis-user' password = 'thesis-password' connect_timeout='2'
[2019-12-09 12:24:18.554] [info] Connected to database
[2019-12-09 12:24:18.554] [info] Checking existence of table in db: phish_score
[2019-12-09 12:24:18.554] [info] OK
[2019-12-09 12:24:18.554] [info] Checking URL in the cache
[2019-12-09 12:24:18.558] [info] URL not found
[2019-12-09 12:24:18.558] [info] Checking existence of table in db: phishtank
[2019-12-09 12:24:18.559] [warning] Table does not exist: phishtank
[2019-12-09 12:24:18.559] [info] Checking URL in Google Safebrowsing
[2019-12-09 12:24:18.722] [info] URL not found
[2019-12-09 12:24:18.724] [info] Checking URL through heuristic model
Executing command: /home/astaruch/Documents/code/master-thesis/modules/html-analysis/bin/html-analysis --feat-src-link --feat-ahref-link --feat-favicon-link --output-json --url 'http://google.com'
Executing command: /home/astaruch/Documents/code/master-thesis/modules/model-checker/bin/model-checker --data-json "{\"ahref_link\":0.6666666666666666,\"asn\":0.279362540483315,\"dns_created\":0.0,\"dns_updated\":0.9874332702495809,\"dnssec\":1.0,\"favicon_link\":0.0,\"gtld\":0.0,\"host_length\":0.0,\"path_length\":0.0,\"similar_domain\":0.0,\"spec_chars_path\":0.0,\"spec_chars_query\":0.0,\"spec_keywords\":0.0,\"src_link\":0.0,\"url_length\":0.02040816326530612}"
[2019-12-09 12:24:22.365] [info] Phishing score: 18
{\"score\":18,\"url\":\"http://google.com\"}
[2019-12-09 12:24:22.365] [info] Saving score in database
```

# B List of attachments

- thesis.pdf – full text of this thesis
- source-code.zip – full source of this work. The source code with its history is also available on `https://github.com/astaruch/master-thesis`
- secrets.env – API keys to Phishtank and Google API