



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich



# **Enforcement Bots: Nothing can block us!**

## **Automating website registration for GDPR compliance analysis**

Bachelor Thesis

Patrice Michael Kast

2nd of March 2021

Advisors: Prof. Dr. David Basin, Karel Kubicek  
Department of Information Security, ETH Zürich

---

## Abstract

Since most web service registrations require an email address, users provide the service contact information that can be used to send marketing emails against their will. To prevent sending these unsolicited emails, countries introduced regulations. Namely in the EU, the ePrivacy Directive defines the rules of sending marketing emails and the General Data Protection Regulation (GDPR) defines the consent users must give to subscribe to mailing lists. In order to inspect the compliance of a website by monitoring marketing emails, we have to register for each of these services, and then observe what types of emails the service is sending us.

Because manual registration is a tedious process, in this work we develop and deploy a crawler, which allows us to scale up the compliance analysis. We call this fully automated crawler enforcement bot (*enfbot*). These enfbots are able to detect a registration form, fill it with artificially generated data, submit it, and check whether the registration was successful. As websites intend to block automated registrations, we had to overcome bot detections such as various versions of CAPTCHAs. Enfbots can detect, extract, and pass the CAPTCHA riddle using a third-party CAPTCHA solver. To evaluate the sign-up success rate of these enfbots, we conducted a large-scale measurement analysis by crawling the one million most popular websites from the Tranco domain list. The enfbots successfully registered to 4.0% of websites, which corresponds to 40'000 web pages. As the required computational time was 6.79 years, we had to accelerate the process by parallelization. We deployed the crawler using Docker and with an 80 core server we speeded up the process 60-fold.

---

## Acknowledgement

I would like to extend my sincere thanks to Professor David Basin and the Information Security Group for funding this project as well as the provision of the use-case to develop the complex automated registration process described in this thesis.

I'm deeply indebted to Karel Kubíček, my mentor in this research project who has taught me a lot about data processing with Selenium, programming in Python and working with Docker images and Docker Compose. These technologies were completely new to me at the beginning of this thesis.

I am grateful to have been able to do this work and to have further developed my experience in the field of web crawling and GDPR compliance.

Last, I would like to acknowledge the support as well as the unlimited cloud computing resources provided by the KastGroup GmbH<sup>1</sup>, without which the crawling process of the one million most popular websites of the internet would not have been possible within one month time.

---

<sup>1</sup><https://kastgroup.com/>

---

# Contents

---

<b>Contents</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Legal regulations . . . . .	1
1.2 Automated compliance analysis . . . . .	2
1.3 Challenges . . . . .	3
<b>2 Registration process</b>	<b>4</b>
2.1 Initial state of the crawler . . . . .	4
2.2 Pre-registration scannings . . . . .	5
2.3 Registration page detection . . . . .	5
2.4 Registration form detection . . . . .	5
2.5 Dummy user credentials . . . . .	6
2.6 Input field detection . . . . .	6
2.6.1 Select object . . . . .	7
2.6.2 Radio inputs & checkboxes . . . . .	8
2.6.3 Regex based value generation . . . . .	8
2.7 Bot protection . . . . .	8
2.8 Submitting registration form . . . . .	8
2.9 Two-step CAPTCHA . . . . .	9
2.10 Registration validation . . . . .	9
2.10.1 State keywords detected . . . . .	10
2.10.2 Registration form disappeared . . . . .	10
2.10.3 Redirect detected . . . . .	10
2.11 Data aggregation . . . . .	10
2.12 Keyword-based algorithms . . . . .	10
<b>3 CAPTCHAs</b>	<b>12</b>
3.1 Types of CAPTCHAs . . . . .	12
3.2 Human solving service . . . . .	13

---

3.2.1	Costs . . . . .	14
3.3	reCAPTCHA v2 . . . . .	14
3.3.1	Visible . . . . .	14
3.3.2	Invisible . . . . .	15
3.3.3	Detection . . . . .	15
3.3.4	Solving . . . . .	15
3.4	reCAPTCHA v3 . . . . .	16
3.4.1	Detection . . . . .	16
3.4.2	Solving . . . . .	17
3.5	hCaptcha . . . . .	17
3.5.1	Detection . . . . .	17
3.5.2	Solving . . . . .	18
<b>4</b>	<b>Distributed crawling</b>	<b>19</b>
4.1	Headless browsing . . . . .	19
4.2	Job distribution . . . . .	20
4.2.1	Crawling database . . . . .	20
4.3	Docker host . . . . .	20
4.3.1	Docker volume . . . . .	21
4.3.2	ProtonVPN . . . . .	21
4.4	Docker . . . . .	21
4.5	Docker Compose . . . . .	22
4.6	Optimizations . . . . .	22
4.7	Costs . . . . .	23
<b>5</b>	<b>Registration statistics</b>	<b>24</b>
5.1	Successful registration statistics . . . . .	24
5.2	Unsuccessful registration analysis . . . . .	24
5.2.1	No registration form on website . . . . .	24
5.2.2	Broken websites . . . . .	25
5.2.3	Link keyword detection . . . . .	25
5.2.4	Complex registration process . . . . .	26
5.2.5	Non-EU language . . . . .	26
5.2.6	Malformed input injection . . . . .	26
5.3	Different codebases . . . . .	26
5.3.1	Version 1.0 . . . . .	26
5.3.2	Version 1.1 . . . . .	26
5.3.3	Version 1.2 . . . . .	27
5.3.4	Version 1.3 . . . . .	27
5.3.5	Comparison: version 1.0 vs. version 1.3 . . . . .	27
5.4	Number of emails received . . . . .	28
<b>6</b>	<b>Additional observations</b>	<b>29</b>
6.1	Bot protection mechanisms . . . . .	29

6.1.1	Sign-up form in an iframe . . . . .	29
6.1.2	Multistep submit process . . . . .	29
6.2	CAPTCHA solving service secret keys . . . . .	30
6.3	Comparison: BuiltWith vs. own usage stats . . . . .	30
6.4	File system out of space . . . . .	31
6.5	Crawling monitoring . . . . .	31
<b>7</b>	<b>Related work</b>	<b>32</b>
7.1	OpenWPM platform . . . . .	32
7.2	Automated registration . . . . .	32
7.3	Automated sign-in . . . . .	33
<b>8</b>	<b>Discussion</b>	<b>34</b>
8.1	Future work . . . . .	34
8.2	Conclusion . . . . .	35
	<b>Bibliography</b>	<b>36</b>
	<b>Appendix</b>	<b>38</b>
	Sources . . . . .	38

## Chapter 1

---

# Introduction

---

Based on DataProt 36% of all world spam content consists of advertisement emails<sup>1</sup> of which 80% are ignored in the inbox by the user<sup>2</sup>. Due to personal interests or a complicated unsubscribe process, only 0.17% of users sign out of mailing lists<sup>3</sup> where 43% give as the reason for doing so that they don't recognise the brand or don't remember signing up to newsletters<sup>4</sup>. It is often unknown from which source these companies could obtain user data and if they are in the right to do so. In this thesis, we<sup>5</sup> propose a programmed registration procedure, which is a critical component of creating an automated process to analyse such data flows and law violations on a big scale.

## 1.1 Legal regulations

Such data collection is strictly regulated in the ePrivacy Directive of the EU [5], which describes the usage of email for direct marketing. As a general rule, this requires the prior consent of subscribers (Art. 3). The concrete definition of consent is defined in the European General Data Protection Regulation (GDPR) [4] which would be applicable in the European Union (EU) and the European Economic Area (EEA).

In case a company does not comply with the mentioned directives, a citizen of the European Union may report the privacy violation to legal authorities, which may fine the firms up to 20 million EUR or 4% of their worldwide turnover.

---

<sup>1</sup><https://dataprot.net/statistics/spam-statistics/>

<sup>2</sup><https://www.zettasphere.com/gmail-promotions-tab-inbox-delivery-stats/>

<sup>3</sup><https://www.campaignmonitor.com/resources/knowledge-base/what-is-a-good-unsubscribe-rate/>

<sup>4</sup><https://optinmonster.com/is-email-marketing-dead-heres-what-the-statistics-show/>

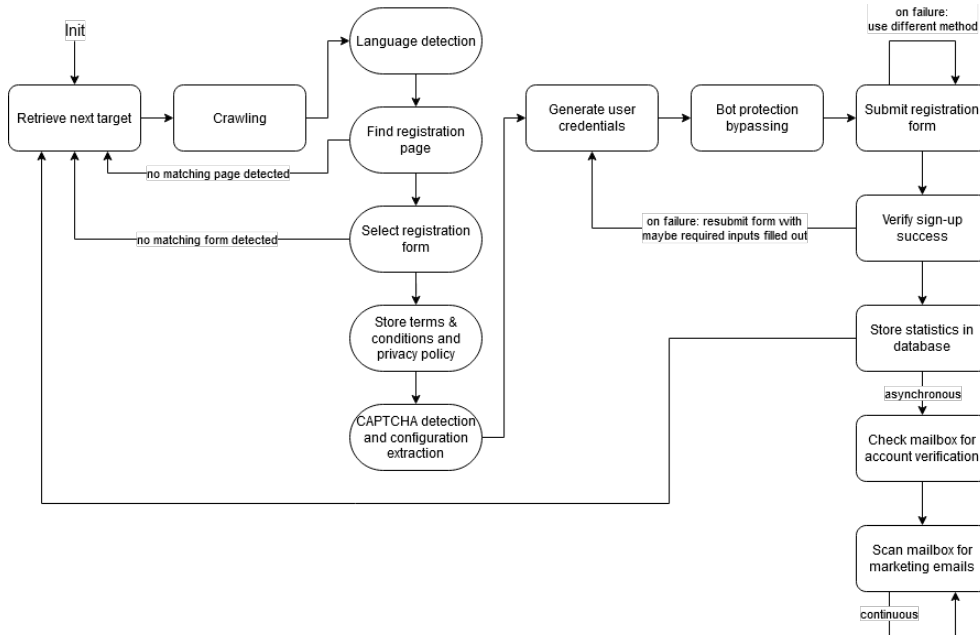
<sup>5</sup>Even though most programming, running the experiments, and writing the thesis text was performed by myself, the plural is used in this thesis. The implementation of the crawler was joined work with Karel Kubíček, the authorship is clarified in Section 2.1.

## 1.2 Automated compliance analysis

While these fines are threatening, the enforcement lacks behind. As a user, the individual damage claims are low in contrast to the high effort to sue websites. To raise awareness for privacy of internet service providers, we propose the usage of enforcement bots, called *enfbots*. They scan the internet on a regular basis and can also be used by administrative regulators like the French CNIL or the English ICO.

To be able to analyse marketing emails on a large scale, we have to store a working email address on the newsletter mailing lists. Doing this manually does not scale. Thus the goal of this thesis is to fully automate the user registration process on websites using the Selenium Framework since user interaction and JavaScript execution is needed to bypass bot detection mechanisms.

We took the Tranco list<sup>6</sup> [7], generated on 11 December 2020, containing the one million most often used websites as a base for our compliance crawling. This domain list was specifically chosen since it is free to use, versioned for reproducibility, and more resistant to popularity manipulations than Alexa top 1M.



**Figure 1.1:** Flowchart of the enfbots with rectangles for states and rounded ones for intermediate steps.

<sup>6</sup><https://tranco-list.eu/list/89WV/1000000>



The flowchart in Fig. 1.1 visualises the internal logic of our fully automated process. The following sequential procedure is initiated on giving an enfbot a specific hostname:

1. crawling a given domain for pages containing a registration form,
2. searching for privacy policy and terms & conditions related to the registration form,
3. generating user credentials for a registration,
4. checking if there is any bot detection mechanism (e.g. CAPTCHAs) in place and bypass these,
5. trying different submission methods on the complete registration form,
6. scanning the website for post-registration bot detection mechanisms,
7. testing for indications of a successful registration with the browser,
8. checking the mailbox for account confirmation emails and clicking on confirmation links, and
9. scanning the mailbox in regular intervals for marketing emails, which may be GDPR violations, as we never gave our consent.

## 1.3 Challenges

Multiple challenges arise when automating this registration process.

1. The keyword link detection does not always work when locating the sign-up form.
2. It is complicated to determine the correct type of each input field.
3. A bot protection mechanism can block the submission of registration forms.
4. Different ways of submitting a sign-up form can improve the success rate.
5. A successful registration might not be correctly validated.
6. Since the large crawl with an unparalleled enfbot would take several CPU years, it has to be speeded up by deploying parallelism.

By an iterative improvement of the source code and many testing rounds, we were able to solve all these challenges and register to 39'343 websites. Although the total computing time was 6.79 years, we completed the crawling within 4 weeks due to distributed computation.

---

# Registration process

---

To analyse the handling of personal data and email notifications by websites, we need to place trackable email addresses in the corresponding newsletter and customer mailing lists. The registration process on these sites would need to be done in a similar way as a typical internet user would do.

Due to the variety of different implementations available, it is not possible to cover all website mailing list registration types. However, as we will demonstrate, it suffices to design a process that works for a representative sample of websites to gather enough data for the following processes. We developed a crawler using a generic sign-up approach that is not tailored to any specific website, such as Jonker et al. [6] have implemented.

### 2.1 Initial state of the crawler

Other members of the team at ETH Zurich already created the base of the crawler as well as the email detection service prior to the beginning of this work. The following components and features were part of the base of the crawler.

***URL input loading*** This provides the initiation of the crawling process of a specific hostname.

***Selenium driver*** This component is used to open a website in a Selenium instance using gekodriver and the Firefox browser.

***Website surfing*** This routine searches for relevant keywords in link HTML-tags to find the registration page, the privacy policy, and the terms of use.

***Form filling*** This procedure fills in all inputs within a registration form using type-detected values.

**Database interactions** These are used to store new websites and to modify related entries.

This given crawler base was significantly extended during this thesis. Besides, the original crawling process was split into two parts; one for automated scraping of website registration forms and the other for the manual annotation work, done by human workers.

## 2.2 Pre-registration scannings

First, the crawler tries to detect the used language on the website with the `polyglot`<sup>1</sup> Python library and uses the value of the `lang`-attribute of the `HTML`-tag as a fallback. The latter was not used as a primary selector since it is often wrongly specified.

## 2.3 Registration page detection

The next step of the automation process is to navigate from the landing page of a website to the registration page. This can be achieved by using keyword detection on all available hyperlinks and searching for key-phrases like *'sign up'*, *'register'*, *'create account'* in English and the detected language on the website. The searchable keywords are translated into 28 European languages since our focus is on European websites and web services targeting European citizens.

In case no direct registration page link could be located, the crawler tries to identify the login page for recurring users. Often, there exists an option or hyperlink to create a new account on these pages. In case no such option is provided to the user, the URL of the login page is scanned for specific keywords which can be replaced with terms like *'create'*, *'join'*, *'registration'* to locate the registration form.

For example, when having the URL `http://example.com/login`, we test if the URL `http://example.com/register` resolves and returns a 200 HTTP-code which confirms that a request has been processed successfully.

## 2.4 Registration form detection

After locating a potential registration page, all available forms are indexed by the number of user input fields such as email- and password-inputs and consequently sorted in decreasing order by the number of total inputs. Besides testing for at least one email input, the indexed form with the highest number of password inputs gets selected for further processing.

---

<sup>1</sup><https://pypi.org/project/polyglot/>

This procedure is required if a registration and a login form are present on the same page. Since a registration form often has more input fields than a simple login form, like a requirement to repeat the new password or the used email address, the probability to select the correct form increases with the above algorithm.

In case a suitable form is detected, the found website is stored in the database and the registration page is marked as a successful hit. Otherwise, it is flagged as Not-Available *N/A* and the enfbots continue crawling other domains.

## 2.5 Dummy user credentials

To be able to assign an incoming marketing email to a specific website, a new unique email address is generated for each registration together with the following user details:

- first-, last- & full-name,
- username,
- street name & number,
- city & postal code,
- country,
- birth-date,
- phone, and
- password.

These values get stored in the database for later input field injection.

The introduced credentials describe a person situated in Germany to be able to test for any GDPR violations later on. However, this artificially generated user is not associated with any real person, neither does the address exist nor is the phone number registered at the point this thesis was written.

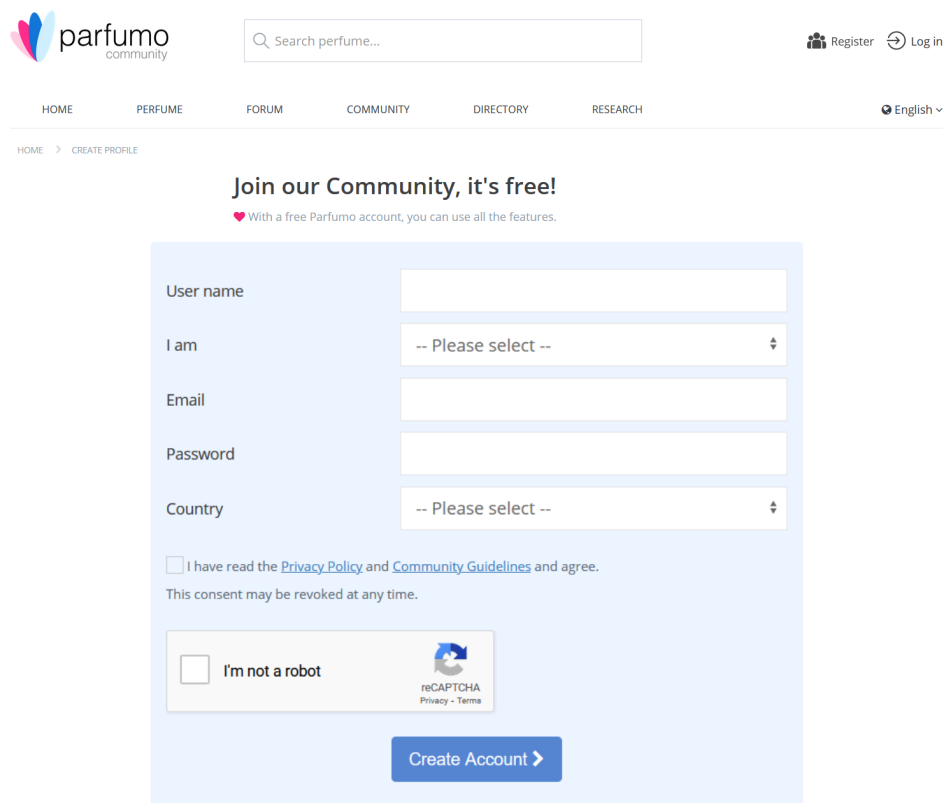
## 2.6 Input field detection

Assuming a suitable registration form was found, all input fields as well as textareas get indexed and tested if they require to be filled in with the corresponding user details. This is done by searching for the HTML required attribute and assessing the input type & class to figure out if it is an essential part of the registration process. This is the case with types like *passwords*, *emails* or *usernames*. Moreover, the enfbots scan the nearest embedded text

## 2.6. Input field detection

next to the input HTML-tag, if it contains an asterisk char indicating necessity.

We use a keyword-based assignment process to fill in the correct user credentials. After filling in all the categorised input fields, the remaining ones are checked for requirement. In case they are, a random value of the correct type is injected.



The screenshot shows the Parfumo community website's sign-up page. At the top, there is a search bar with the placeholder text "Search perfume...". To the right of the search bar are links for "Register" and "Log in". Below the search bar is a navigation menu with links for "HOME", "PERFUME", "FORUM", "COMMUNITY", "DIRECTORY", and "RESEARCH". On the far right of the navigation menu is a language selector set to "English". Below the navigation menu is a breadcrumb trail: "HOME > CREATE PROFILE". The main heading of the page is "Join our Community, it's free!". Below this heading is a sub-heading: "With a free Parfumo account, you can use all the features." The sign-up form itself is a light blue box containing several input fields: "User name", "I am" (a dropdown menu with "-- Please select --"), "Email", "Password", and "Country" (a dropdown menu with "-- Please select --"). Below these fields is a checkbox for "I have read the Privacy Policy and Community Guidelines and agree." with a link to "Privacy Policy" and "Community Guidelines". Below the checkbox is a link: "This consent may be revoked at any time." At the bottom of the form is a reCAPTCHA widget with the text "I'm not a robot" and a "Create Account" button.

**Figure 2.1:** example sign-up form on parfumo.de

Figure 2.1 shows an example of a typical sign-up form containing a select HTML-tag for gender selection as well as one for the country. To submit the form, the terms & conditions checkbox has to be accepted and the riddle of the reCAPTCHA has to be solved.

### 2.6.1 Select object

To deal with select objects which often are required (e.g. users gender or country of residence), the enfbots scan their options to select the first

item with non-whitespace value or `innerText` and which does not possess a `disabled` attribute.

### 2.6.2 Radio inputs & checkboxes

In contrast to the `select` objects, the crawler deals with radio inputs in a far less sophisticated way, since their usage is non-uniformly and often one option is as good as any other. For each name related to radio HTML-tags, it is searched for the first occurrence without the `disabled` attribute within the sign-up form which is then automatically `clicked()`.

Checkboxes on the other hand are automatically selected in case they are detected to be required, based on the assessment explained in Section 2.6.

### 2.6.3 Regex based value generation

The `pattern` attribute was introduced in HTML 5, which allows a website to ask the browser for custom input validation, based on regular expressions.

After the keyword-based input generation is completed, it is checked if the input is `type=text` and contains a `pattern` attribute. If this is the case, the reverse regex function from the `rstr` Python library is used to generate a matching value to the input field. This process is only performed on inputs of type `text` since we want to prevent the replacement of any better matching values and identifiers like email addresses.

## 2.7 Bot protection

Azad et al. [1] showed that at least 13.6% of the Alexa 1M websites use some sort of bot protection mechanism to prevent bots from creating accounts, whilst based on BuiltWith, 19% of all websites are using a CAPTCHA system<sup>2</sup>. Since CAPTCHA-systems can be instantiated and configured in different ways, we will explain the further solving process in detail in Chapter 3.

## 2.8 Submitting registration form

After filling the form with the generated user credentials and during solving the CAPTCHA, a screenshot of the website is taken for further analysis. Additionally, the terms & conditions and the privacy policy are stored in an HTML-file.

Since we need to submit the form we filled in to place our email in the mailing list of the website, the submission process is essential. During testing, it was discovered that the default `submit()` function of Selenium is not

---

<sup>2</sup><https://trends.builtwith.com/widgets/captcha>

working on every discovered form. The reason for this behaviour could not be located. By reverse-engineering the Selenium library, the corresponding JavaScript code was found:<sup>3</sup>

```
1 driver.execute_script('arguments[0].submit();', obj)
```

Since this code is Vanilla JavaScript, we were able to prevent failure exceptions by testing the type of the function to be executed. Furthermore, it was brought to light that in the other cases, where the code above generates faults, the `submit()` function has to be called as a `click` event on the submit object. Thus, the following two lines were implemented to support the submission of the prepared registration form:

```
1 browser.driver.execute_script('if(typeof arguments[0].submit === "function"){  
  ↪ arguments[0].submit(); }',obj)  
2 browser.driver.execute_script('if(typeof arguments[0].submit.click ===  
  ↪ "function"){ arguments[0].submit.click(); }',obj)
```

In later tests, it was even discovered that calling the `submit()` function on registration forms directly may trigger an empty page reload, probably implemented as bot protection. For this reason, an algorithm was written which first tries to click on buttons and `input[type=submit]` before calling `submit()` on the form. In case a candidate button was found, it gets tagged with a custom class which is later used as an indication, that the `click()` event actually submitted the form. Otherwise, the fallback method from above is used by submitting the form directly.

## 2.9 Two-step CAPTCHA

Some websites display a CAPTCHA to verify an account creator is human just after the form submission. This is due to some price savings since website administrators are charged with each spawning of a CAPTCHA. By delaying the instantiation post the registration form submission, costs can be cut down. To complete the automated registration process, the crawler has to check for any signs of an active CAPTCHA system that tries to prevent the registration by a bot. If it found any, the solving process of the CAPTCHA is repeated.

## 2.10 Registration validation

In case the registration form could be submitted and no two-step CAPTCHAs could be found or if they were already solved, the success of the registration has to be validated. This can be achieved as explained in the following sections.

---

<sup>3</sup>The embedded code examples in this thesis are partly substituted for better readability.

### 2.10.1 State keywords detected

To detect a registration state with redirects, we also scan the text of the HTML for keywords that indicate a failure or a success as the result of the automated process. These keywords are verified not to be existent before the form submission, to make sure no text which is already embedded on the page beforehand is matched. The content is then extracted from the HTML-code by removing the tags and ignoring the inner values of tags like `script`, `header`, `style` or `meta`.

### 2.10.2 Registration form disappeared

After a successful registration the form often completely disappears. If we detect the same form as before the registration attempt or fetch "non-success-indicating keywords" in the response, it can be assumed that the registration failed due to some false value-injections on the inputs or not checking all mandatory checkboxes.

### 2.10.3 Redirect detected

In case the user is redirected to a URL that fulfils the following requirements, the registration is regarded as successful.

*Different hostnames* indicate the redirect of the user to another subdomain.

*Different paths* imply the relocation of the user within a site.

These conditions were defined based on observations during tests with a few hundred domains. The query part of the URL is not considered intentionally, due to some malformed form submission over GET, which submits the user inputs as unsafe query parameters to the server.

## 2.11 Data aggregation

After the crawling, data aggregation is performed with two storage locations, one for saving files and the other for structured data. All information extracted from the website is stored in a central database that contains cookies, credentials used for sign-up as well as CAPTCHA extractions. Screenshots of the filled-in registration form as well as terms & conditions and privacy policies HTML source codes are stored in the file system in an output directory for later bug tracing and analytics.

## 2.12 Keyword-based algorithms

All mentioned keyword-based algorithms in this thesis try to use words in the detected language of the website as well as in English as possible



fallback values. After starting just with English and German translation, we decided to automatically translate our wordlists into every major national language which is spoken in a country within the EU and thus under GDPR protection.

Table 2.1 shows the translated languages used together with crawling statistics of how many websites were found in the given language.

English (en)	37.38%	Latin (vai)	0%
German (de)	3.61%	Latvian (lv)	0.04%
Albanian (sq)	0.02%	Lithuanian (lt)	0.03%
Basque (eu)	0.01%	Luxembourgian (lb)	0.001%
Bosnian (bs)	0.01%	Macedonian (mk)	0.01%
Bulgarian (bg)	0.08%	Maltese (mt)	0.001%
Catalan (ca)	0.05%	Norwegian (no)	0.10%
Croatian (hr)	0.05%	Polish (pl)	0.64%
Czech (cs)	0.30%	Portuguese (pt)	0.71%
Danish (da)	0.13%	Romanian (ro)	0.17%
Dutch (nl)	0.74%	Russian (ru)	3.53%
Estonian (et)	0.02%	Serbian (sr)	0.06%
Finnish (fi)	0.10%	Slovak (sk)	0.10%
French (fr)	1.60%	Slovenian (sl)	0.03%
Galilean (gl)	0.013%	Spanish (es)	1.67%
Greek (el)	0.24%	Swedish (sv)	0.19%
Hungarian (hu)	0.17%	Turkish (tr)	0.73%
Icelandic (is)	0.01%	Ukrainian (uk)	0.17%
Irish (ga)	0.002%	Welsh (cy)	0.003%
Italian (it)	0.64%		

**Table 2.1:** Used languages for keyword-based detections

On a total of 373'852 websites, the detected language was not of European origin and thus the website was not further analysed. In addition, it has to be noted that the user agent we used to access the websites indicated English as the desired website language. Thus, these numbers may not correspond to the native language of these websites if they additionally support English as an option.

With Brexit, the UK will not be part of the EU any longer. Nevertheless, it will have similar data protection levels as it was confirmed on the 19th of February 2021, by the European Data Protection Board in an adequacy decision.<sup>4</sup>

<sup>4</sup>[https://ec.europa.eu/info/law/law-topic/data-protection/international-dimension-data-protection/brexit\\_en](https://ec.europa.eu/info/law/law-topic/data-protection/international-dimension-data-protection/brexit_en)

## Chapter 3

---

# CAPTCHAs

---

To prevent bots from registering to email or newsletter services, many websites deploy some sort of bot detection mechanism. The most efficient way of achieving this is by asking the interacting user to solve a riddle, a so-called CAPTCHA, which complicates the automated process of registration as described in this thesis.

CAPTCHA was originally designed for academic studies by Von Ahn et al. [9], but over the last few years, some CAPTCHA-systems became quite popular and are widely used today. In the following sections, the most popular CAPTCHA-systems get introduced and it is explained how to bypass them. The information concerning their popularity is taken from BuiltWith, an internet crawler that detects software components integrated into websites.<sup>1</sup>

### 3.1 Types of CAPTCHAs

After consulting the usage distribution of diverse implementations, we decided to focus our work on the following most often used CAPTCHA-systems based on BuiltWith.

***reCAPTCHA v2 Visible*** (6% of the one million most popular websites<sup>2</sup>)

The reCAPTCHA v2 system was designed by Google and displays a riddle to the user, which has to be solved prior to the form submission.<sup>3</sup>

---

<sup>1</sup><https://trends.builtwith.com/widgets/captcha>

<sup>2</sup><https://trends.builtwith.com/widgets/reCAPTCHA-v2>

<sup>3</sup><https://developers.google.com/recaptcha/docs/display>

**Invisible reCAPTCHA** (similar usage to reCAPTCHA v2, see Footnote 2)

In contrast to the visible system, the invisible one is only displayed to the user in case the probability of being a bot is high.<sup>4</sup>

**reCAPTCHA v3** (3.5% of the one million most popular websites<sup>5</sup>)

The reCAPTCHA v3 plugin is a further development of Google and the successor of v2. It is almost completely server-side and rates a specific user with a bot score in the interval from 0.0 to 1.0.<sup>6</sup>

**hCaptcha** (0.0375% of the one million most popular websites<sup>7</sup>)

The hCaptcha was created to provide a better privacy data protection to the user in contrast to the reCAPTCHA-systems.<sup>8</sup>

Neither the most famous image CAPTCHA model where the user is given an image and has to enter a number nor the math CAPTCHA model is asked by the CAPTCHA-systems above. The reason we did not implement a solving strategy for the image CAPTCHA was that there does not exist one popular uniformly used CAPTCHA-system. Since almost all of these systems have individual implementations, we would not have the resources to cover them for our work. Thus this task was postponed to be solved in future work as mentioned in chapter 8.1.

Based on BuiltWith, less than 0.3% of websites containing a CAPTCHA are not solvable by the enfbots since another CAPTCHA-system is implemented. The most basic version of displaying an image or math CAPTCHA directly within a form and only use server-side routines to verify the user's answer may not be detected by BuiltWith and thus is not included in the 0.3%.

## 3.2 Human solving service

Since the project aims to design a fully automated registration process, the CAPTCHAs can not be solved by hand. Thus, a capable CAPTCHA solving service had to be found.

Multiple machine learning based approaches were tested but none of them delivered reasonable results when it came to solving a modern CAPTCHA-system like the one described in this chapter. An exception to this statement are the services that state on their website that they use machine learning for the solving process, but are clearly based on human-computing power. The following human solving services were further assessed:

---

<sup>4</sup><https://developers.google.com/recaptcha/docs/invisible>

<sup>5</sup><https://trends.builtwith.com/widgets/reCAPTCHA-v3>

<sup>6</sup><https://developers.google.com/recaptcha/docs/v3>

<sup>7</sup><https://trends.builtwith.com/widgets/hCaptcha>

<sup>8</sup><https://www.hcaptcha.com/>

- AntiCaptcha<sup>9</sup>,
- AZCaptcha<sup>10</sup>, and
- ImageTyperz<sup>11</sup>.

After testing their capabilities by providing samples to solve, we finally decided to use the ImageTyperz service which never returned a false-solved CAPTCHA riddle response or a timeout during the evaluation.

Other providers did sometimes fail at the given tasks and also wrongly stated on their website that they would rely on machine learning to solve the CAPTCHAs.

The specific ways of detecting and solving these CAPTCHA types using the selected ImageTyperz service is explained in detail in the next sections.

### 3.2.1 Costs

Crawling the one million of most popular websites resulted in a total of 22'524 successful solving-requests which is equal to USD 35.76\$. Even though the enfbots did not detect any CAPTCHA related errors, ImageTyperz stated a total of 5'380 timeouts on their dashboard. We were not able to detect these because we set the maximum crawling time of one page to 180 seconds in the enfbots, which may be below the timeout value of the CAPTCHA solving service.

## 3.3 reCAPTCHA v2

Googles reCAPTCHA v1 was developed back in 2009. While the original version is no longer supported, their v2 service is the most implemented CAPTCHA-system worldwide.<sup>12</sup>

The two ways of implementing the reCAPTCHA v2 system are visible and invisible, further described below.

### 3.3.1 Visible

To protect from bots, the easiest and most often used way to instantiate the reCAPTCHA is by embedding it directly within the HTML-form. Prior to submitting any information, the user has to verify his humanness by clicking a checkbox called "I am not a robot". In case the reCAPTCHA algorithm detects anomalies during that process, the user is given a keyword

---

<sup>9</sup><https://anti-captcha.com/mainpage>

<sup>10</sup><https://azcaptcha.com/>

<sup>11</sup><http://www.imagetyperz.com/>

<sup>12</sup><https://trends.builtwith.com/widgets/reCAPTCHA>

and matching images to select. Only if the user succeeds in this test, he is allowed to finally submit the form. The backend of the website receives a confirmation through an API call to Google, that the user is not a bot.

### 3.3.2 Invisible

In contrast to the visible reCAPTCHA v2, the invisible reCAPTCHA is not shown to the user with exception of the privacy banner "protected by reCAPTCHA". Again, the user interaction is monitored before the form-submission and in case of suspicion, a similar riddle like in the visible version would be displayed.

### 3.3.3 Detection

To detect a reCAPTCHA v2, a target website is scanned for a script tag containing `/recaptcha/api.js` path in the link. Next it is searched for a custom `onload()` function, a `render=explicit` or `render=hidden` parameter in the source code of the site. Since these options are only used in reCAPTCHA v2, the CAPTCHA-system of the website can be identified and classified accordingly. Additionally, in case the script tag does not fulfil the conditions to be reCAPTCHA v3 (see chapter 3.4), the v2 is used as a fallback type.

### 3.3.4 Solving

As soon as the user response is accepted, the visible and the invisible type reCAPTCHA v2 inject a secret token payload into a hidden HTML input element which is then sent to the server. At the backend, the website can query API endpoints from Google if the token is valid and the user is classified as not being a bot.

Since there is no validation of whether the IP of the user solving the reCAPTCHA and the IP where the form submission originates are identical, it is possible to outsource the solving of the riddles to specialised services using human workforce.

In order to enable the external service to solve the reCAPTCHA for our automated crawling process, the `sitekey` of a website is extracted and sent along with the solve request. This key is often found as an attribute in a `div` HTML-tag containing `class="g-recaptcha"` like the following one:

```
1 <div class="g-recaptcha" data-sitekey="XXXXXXXXXX"  
  ↪ data-callback="submit_form"></div>
```

The `sitekey` can also be defined in a custom `onload` callback function, which is called by the reCAPTCHA plugin after loading. This function would be declared in the `api.js` script tag, which links to the reCAPTCHA JavaScript plugin.

```
1 <script src="https://www.google.com/recaptcha/api.js?onload=olC">
2 </script>
3 <script type="text/javascript">
4     var olC = function() {
5         grecaptcha.render('html_element', {
6             'sitekey' : 'XXXXXXXXX',
7             'callback' : 'submit_form'
8         });
9     };
10 </script>
```

After receiving the required response from the human solving service, the token payload is automatically injected into the hidden HTML-textarea with `id="g-recaptcha-response"` which originally was created by the reCAPTCHA plugin within the registration HTML-form on page load.

Some websites add custom flags or processes to the HTML-form by defining a reCAPTCHA callback function which is executed on successful riddle solving. This function would be defined either within the same HTML-object or the onload callback function as the sitekey was found. A detected custom callback routine is executed prior to submitting the form.

To get the content of the custom callback function from the JavaScript files we use regular expressions due to the applied encapsulation. In contrast, the extraction from the HTML-attribute works with a simple getter method.

## 3.4 reCAPTCHA v3

The main difference of reCAPTCHA v3 as the successor of the reCAPTCHA v2 system is that the newer version does not use any user interaction. A JavaScript-tag must be existent in the head section of the HTML-code to analyse the interactions of a user with the browser. So-called actions allow the reCAPTCHA v3 system to protect different HTML-forms on the same page. These actions need to be included in every form for later reference.

After the submission of the registration HTML-form, the recorded user behaviour is analysed and a score from 0.0 to 1.0 gets calculated which then indicates the probability of the user being a human. This rating can be retrieved via an API call by the backend of the website to decide whether a registration request should be granted or declined.

### 3.4.1 Detection

The reCAPTCHA v3 is implemented on a website by inserting a script tag sourcing `/recaptcha/api.js` in a similar way as it was done in the older v2 system. The JavaScript will instantiate the new system by the time the sitekey is handed over within the render attribute. By the collection of

testing data received through web crawling, we discovered that the sitekey has an exact length of 40 chars. Thus, if a render attribute is detected in the script HTML-tag with the corresponding length, the system is classified as reCAPTCHA v3.

```
1 <script src="https://www.google.com/recaptcha/api.js?render=XXXXXXXXXX">
```

Sometimes the presence of this CAPTCHA-system can also be revealed by testing if the `__greaptcha_cfg.clients` JavaScript object is existent and represents a value greater or equal 10.

```
1 if exec_js("if(typeof __greaptcha_cfg != 'undefined') {  
  ↪ __greaptcha_cfg.clients }") and int(  
2   exec_js("Object.keys(__greaptcha_cfg.clients)[0]")) >= 10:  
3   captcha_type = CaptchaType.ReCAPTCHA3
```

### 3.4.2 Solving

One way of solving this CAPTCHA is to retrieve the sitekey as well as the correct action parameter, located as an attribute in the `class="g-recaptcha"` HTML-tag. Afterwards, the human solving service is requested for a user-token that originates from a user with clean browser history and human-like website interactions. The result is used to replace the user-token within the hidden `id="g-recaptcha-response"` HTML-textarea, which was generated on loading the website. By submitting the request with this new user-token we achieve that the registration is done in the name of a trustworthy non-bot user and will thus be accepted. The difficulty is to reach a score level that is accepted by the website.

## 3.5 hCaptcha

Quite similar to reCAPTCHA v2, a user who wants to submit an HTML-form protected by hCaptcha must solve a riddle. This can again be done by selecting the correct images matching a given keyword. As stated on their website, the improved privacy aspects concerning tracker cookies are the biggest difference to Google's reCAPTCHA.

### 3.5.1 Detection

To protect an HTML-form on a website with the hCaptcha-system, a script tag containing `src="https://hcaptcha.com/[version]/api.js"` has to be included in the head section of the HTML-code. In addition, a HTML-tag `class="h-captcha"` has to be inserted within the form. If we are able to locate these requirements on a website, the CAPTCHA is ultimately classified as hCaptcha.

### 3.5.2 Solving

Like in the other CAPTCHA-systems, a website is identified using an embedded sitekey in the `class="h-captcha"` HTML-tag. This key is forwarded to the human solving service for answering the given riddle and returning the secret-token. We then need to inject the token into a hidden `name=h-captcha-response` HTML-textarea that finally is submitted along with the registration request.

```
1 <div class="h-captcha" data-sitekey="XXXXXXXXX"></div>
```



---

# Distributed crawling

---

Small tests prior to the planned one million domain crawl indicated an immense amount of computation time needed to run the Selenium browser, later determined to be 6.79 years. Therefore, it was decided to convert the complete crawling setup to Docker containers. By using Docker Compose, the number of running containers had been massively scaled up and parallelisation became possible. In comparison to running a big number of virtual machines, Potdar et al. [8] showed that Docker containers generate much less overhead. Another benefit was the circumstance that these containers are much faster deployable which significantly contributed to the development.

## 4.1 Headless browsing

To reduce the memory usage for the required parallelisation on the enfbots, the Selenium browser needed to run headless. Even though Firefox supported a `--headless` option, crawler designers as Zeber et al. [10] are worried that this option may be detected easily by websites using browser fingerprinting. More sophisticated techniques were pulled from Englehardt et al. [3].

Since we do not want to be classified as bots, we have taken the advantage of the `pyvirtualdisplay`<sup>1</sup> Python package in combination with `Xvfb`<sup>2</sup>. By using these libraries which draw the graphical interface of the browser inside the virtual frame buffer the desired functionality was reached. This buffer allows to take screenshots of the filled out registration form for a later retrace of failures during the crawling. This approach of running the browser headless was also used in the OpenWPM project (chapter 7.1).

---

<sup>1</sup><https://pypi.org/project/PyVirtualDisplay/>

<sup>2</sup><https://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml>

## 4.2 Job distribution

The existing base of the crawler provided the following simple parameters for controlling the crawler:

- u: path to the file containing the domains to scrap,
- f: index in domain list to start scrapping from, and
- t: index in domain list to stop scrapping.

To achieve a balanced crawling job distribution with parallel workers, a static URL list was not flexible enough. Instead, the `jobs` table was added to the database, containing all the domains to crawl and the `start_date` and `end_date` of the related crawling process.

Through the start and end information, it was possible to recrawl a domain on an unexpected failure of the enfbots. Jobs which have a `start_date` older than two hours and an `end_date` still set to `NULL` are cleared for reprocessing. Each enfbot selects a random non-finished domain from the available jobs repeatedly until all entries are crawled.

### 4.2.1 Crawling database

To control the complete crawling processes as well as to store the structured data, a PostgreSQL database is used which is handled by the Python object relation-mapping package `Pony ORM`<sup>3</sup>. This component was first located at servers of the ETH. Due to firewall configurations which posed an issue to management and accessibility from outside the ETH network, a different approach had to be taken.

Since the Docker host was already located inside a private data center and therefore not within the ETH network perimeter, the decision to move the database to a cloud-based hosting service seemed a good option to mitigate the problem.

## 4.3 Docker host

We first tried to run the Docker host on a CentOS 8 operating system. Since the development of the source code for the enfbots was done using an Ubuntu virtual machine, running on CentOS 8 resulted in many failures. Thus we decided to base the Docker Host on an Ubuntu 20.04 Image. Intentionally, a complete Desktop Ubuntu distribution was installed to feature an easier installation and maintenance, since the X server is installed by default and can be enabled or disabled.

---

<sup>3</sup><https://ponyorm.org/>

As the ETH supercomputer does not provide any internet access by policy, the Docker Host was set up on a virtual machine running on a private Proxmox<sup>4</sup> cluster to overcome this limitation.

### 4.3.1 Docker volume

On the Docker Engine, a Volume named `crawling_output` was created as a persistent storage for the scraped terms & conditions, privacy policies, and the screenshots taken by the enfbots.

### 4.3.2 ProtonVPN

To load websites exactly in the way they would be presented to a user from the European Union (where the GDPR is applicable), a paid VPN service was subscribed from ProtonVPN<sup>5</sup> which auto-connected at boot time of the Docker Host VM. At the first attempt, it was discovered that the database traffic was not able to reach the crawling database through the proton0 VPN interface. Thus, a startup script was installed to create a route over the direct `ens18` interface of the virtual machine, which is not routed over VPN.

## 4.4 Docker

The individual Docker containers were built based on a Dockerfile. A simpler version already existed in the source code prior to this thesis.

The Dockerfile is based on Ubuntu 20.04 and instructs the Docker Engine to combine the following tasks:

- copy the complete python3 source code into the container,
- install all required software like python3, Firefox, Xvfb, npm or Node.js,
- use pip to fetch the required python3 packages for the advanced crawling process, and
- prepare the execution environment by declaring the entry bash script for starting the crawling as well as defining environment variables.

The most important property of Docker is its easy deployment as well as the ability to be run by anyone, independent of the hardware specifications. An additional benefit is that the results of this study can be reproduced as required in a simple way.

---

<sup>4</sup><https://www.proxmox.com/>

<sup>5</sup><https://protonvpn.com/>

## 4.5 Docker Compose

The written Dockerfile could be used in combination with Docker Compose, a service which can connect the central Docker Volume as well as the Docker Network to containers without specifying it on every execution in the command line.<sup>6</sup> Using the `restart` condition, the Docker Engine can be instructed to keep the containers running at all time, also on restart of the host VM. On starting the Docker Container with `docker-compose up --scale enfbot=n --detach`, the enfbot container can be scaled up by  $n$ -times for  $n$  concurrent running enfbots.

## 4.6 Optimizations

The docker host virtual machine was initially equipped with 16 kvm64 assigned cores and 30 GB of RAM to run 10 Docker containers. The desktop mode of Ubuntu was disabled by updating the grub loader, which freed 1 GB of RAM.

Subsequently, the crawling performance of different configurations of attached resources and the number of concurrent Docker containers was measured over a period of 6 hours. The result is shown in Table 4.1.

Cores	RAM (GB)	Containers	Domains per day
16	30	10	4'488
16	70	30	11'400
24	50	30	16'800
80	100	30	31'604
80	400	30	31'700
80	400	45	40'832
80	100	60	44'320
80	400	60	57'142

**Table 4.1:** Crawling performance

The most performance limiting hardware component was the number of assigned CPUs to the VM, while the RAM was never fully utilised. Since the Proxmox cluster only supported virtual machines with up to 24 cores, the Docker host had to be transferred to a single, more-powerful hypervisor which allowed a scale up to 80 Cores. The new hypervisor specifications were 80 x Intel(R) Xeon(R) CPU E7-8870, 512 GB RAM running Proxmox 6.3-2.

<sup>6</sup><https://docs.docker.com/compose/>

With the scaled up hardware resources and by completing the crawl, we discovered that the most limiting component was the available network bandwidth that was restricted to 150 Mbit/s due to a misconfiguration. Another limitation was identified as storing all the data in one single output directory which also caused a slow-down on progressing in the large crawl.

Finally, we learned from our measurement that the best configuration for the Docker setup is to allocate at least one processor for each container together with approximately 10 GB of RAM and 5 GB of disk space.

## 4.7 Costs

Since we received the computational resources used in this thesis without charging the costs, we rely on publicly available information of Microsoft Azure and on Amazon Web Service (AWS) to estimate the costs incurred by the crawling process of the one million most popular websites. Our configuration for this assessment was as follows:

- Ubuntu 20.04,
- 80 Processors (CPU),
- 400 GB of RAM,
- 600 GB of Diskspace,
- computation-time of 34 days, and
- dedicated host / on-demand.

These requirements resulted in an average estimated cost of USD 5'039\$ for the complete crawling process or 0.50 cents per processed domain if we would go with dedicated hardware. In case we switch to Spot instances and accept the small overhead, on AWS it may be as cheap as USD 1'132\$ or 0.11 cents per processed domain.

# Registration statistics

---

It took a total of 30 days to complete the whole top 1M Tranco domain list, with 4 days of interruption due to unexpected failures. Out of the 1'000'000 websites, the registration process verification resulted in 39'343 successfully created accounts. The crawl collected screenshots and page sources of all visited websites, a total of over 4 million files, taking 137 GB of storage.

In this chapter we present the results and inspect the failing registrations, showing which of the individual steps caused the failures.

## 5.1 Successful registration statistics

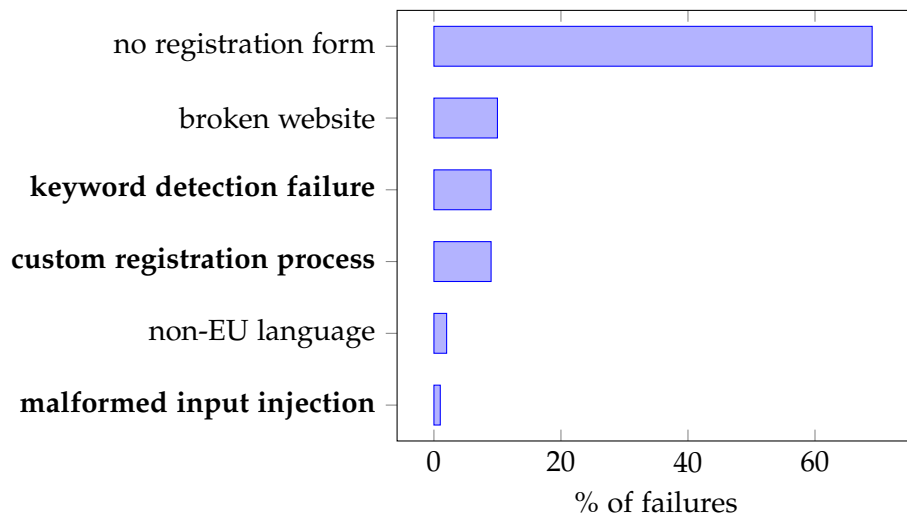
Our enfbots managed to register to 3.9% (39'343) of crawled websites. This was determined by our register verification techniques as described in Section 2.10. Of these successful registrations, 9.7% (3'801 websites) used a CAPTCHA which the enfbots detected and successfully passed. On the other hand, we detected a CAPTCHA-system on 1.6% of all websites (16'308) of which only 23.3 % led to a verifiable success of the followed registration.

## 5.2 Unsuccessful registration analysis

To improve the enfbots, we manually analysed 100 websites chosen randomly where our automated registration failed. The assessment that we summarised in Fig. 5.1 provided useful insights which we categorised into the following 6 types of failures on which we elaborate below.

### 5.2.1 No registration form on website

69 of 100 failed websites did not contain a way to sign up or create a new account as the enfbots correctly determined. Of these, 10 target websites did only contain a login form for a closed community whilst the others did not



**Figure 5.1:** Distribution of registration failure sources. The highlighted labels mark an error related directly to the enfbots.

have an account system implemented at all. The non-existence of a registration form explained the failure to detect the input fields and therefore the failure was not related to the crawling process of the enfbots.

### 5.2.2 Broken websites

In ten of the 100 analysed samples, the HTTP request either resulted in a DNS failure, a timeout, or contained the directory listing of the web folder in the HTTP result. This behaviour indicates that the Tranco domain list is not up-to-date and websites were collected over a longer time. Again, no failure was related to the enfbots.

### 5.2.3 Link keyword detection

Since we used keyword detection to browse a page for the registration form, it was expected to be the most error-prone component of the crawling process. This was confirmed since nine of the analysed failed registration attempts could be led back to neither detecting a registration- nor login-related keyword in the HTML tags.

Most often, the reason for not detecting the correct login-path was the usage of icons instead of text to navigate a user through the website. We address this issue in version 1.3 of the codebase, explained in Section 5.3.4.

#### 5.2.4 Complex registration process

In nine cases, the failures could be combined to the general problem of not being able to detect a custom registration process on the website. Two websites did not allow a registration without completing an order, two hid their content behind a landing page, two gave a user multiple options of creating different types of accounts, and in three cases the sign-up form was instantiated using JavaScript with a delay after the page load, where the enfbots did not wait long enough. To mitigate this issue, JavaScript procedures in the Browser could be tracked or a larger sleeping-time could be implemented after a page load before the detection process starts.

#### 5.2.5 Non-EU language

In only two of the failed registration attempts we analysed, the enfbots correctly detected non-EU languages (Vietnamese and Japanese in the analysed samples). Since these websites are clearly not targeting users originating from the EU, this source of failure can be ignored. On these websites the enfbots just tried to register using English keywords as fallback.

#### 5.2.6 Malformed input injection

In one observation the error could be traced back to an incorrect input value generation, where the website did not provide a regex pattern for a specific input field whose value was checked afterwards using JavaScript. Since the expected input was the BMI of a user, our keyword detection algorithm had no chance of categorising it correctly.

### 5.3 Different codebases

In this section, we describe the incremental improvements to the crawling process and compare two presented versions on a random sample of 10'000 websites.

#### 5.3.1 Version 1.0

We label the first version of the source code capable of register on CAPTCHA protected websites 1.0. This code was also used for the scaled crawling. While the statistics in this chapter relate to this code, we propose several optimisations based on our observations.

#### 5.3.2 Version 1.1

We noticed that the naive strategy of selecting the first non-disabled option of a `select` HTML-tag was the reason why Afghanistan was always chosen



as the country of our fictitious user, as it is alphabetically the first country available. Since Afghanistan is a non-EU region and thus the GDPR may not apply to user data entered on the websites, we decided to implement a keyword-based linear search through all non-disabled options in expectation to detect a Germany-residence option. While this does not directly improve the registration rate, it enhances our abilities to detect privacy violations.

### 5.3.3 Version 1.2

In this version, we tried to improve the completeness of the form inputs, by filling more than the determined required input fields. In addition to checking for a required attribute or asterisk character, the text of checkboxes is also tested for marketing related keywords. When none of the marketing keywords has been detected, it is searched for other keywords indicating terms & conditions or a privacy policy and if found, we mark this HTML-input as *maybe-required*. This keyword mapping process is implemented in two steps, since often multiple checkboxes are described using the same text object, where they are embedded in, and thus we would risk subscribing to any optional mailing lists as well. To further minimise this risk, the registration form is submitted by filling in only required inputs on the first attempt and if the same form is re-detected after the page-reload, also maybe-required checkboxes are checked.

### 5.3.4 Version 1.3

As explained in Section 5.2.3, a big percentage of websites rely on icon-buttons instead of text-links with the result that the correct registration page could not be located with our keyword-based approach. Therefore we propose the scanning for `alt` and `title` HTML-attributes on all link HTML-tags and their children, since these attributes are often used to allow navigation through the website for blind people.

In addition, we followed the recommendations by Jonker et al. [6] who noticed that iterating over many HTML elements using the Selenium API is significantly slower than an equivalent executing with Vanilla JavaScript. So we created the keyword crawler of `alt` & `title` HTML-attributes directly in JavaScript.

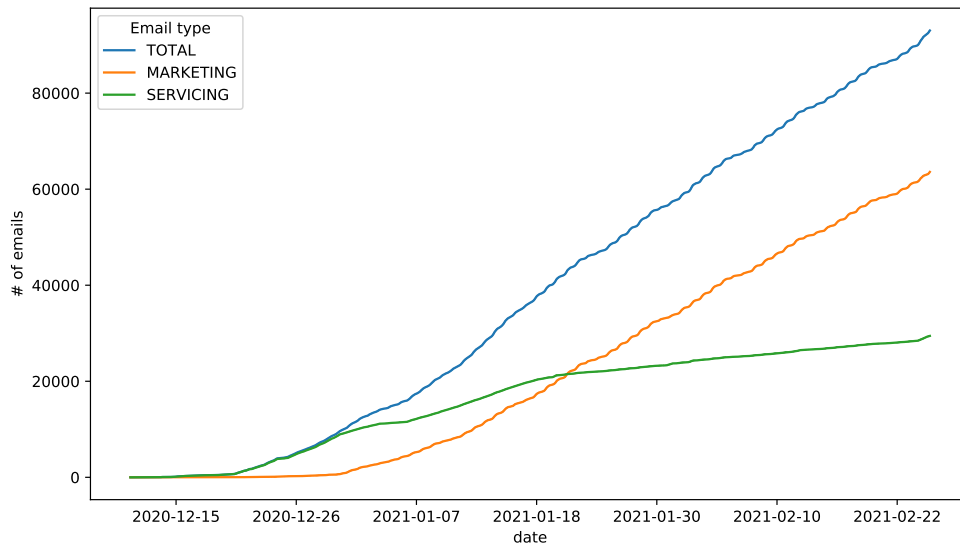
### 5.3.5 Comparison: version 1.0 vs. version 1.3

By rebuilding the Docker containers based on version 1.3 of the source code, we recrawled 10'000 websites to be able to compare the registration successes. After a total crawling time of 5 hours instead of 7.2 hours, the enfbots achieved a registration success on 4.0% of the websites, which is a slight improvement compared to the 3.9% success rate of version 1.0. This is

equivalent to 1'000 additional websites when scaled up to the whole Tranco domain list.

## 5.4 Number of emails received

Throughout the iterative testing rounds, we received 89'851 emails. Fig. 5.2 shows the number of received emails since the beginning of the large crawl. Of the 89'473 emails, only 86'571 were sent to addresses created for the large crawl. The remaining 2'902 were received by addresses we used for iterative improvements of the crawler. We received at least one email from 6'779 websites of the 39'343 webpages where we registered successfully. Although it is expected that many websites do not communicate with the user unless he would need to reset his password, this low communication rate suggests that our detection of successful registrations may be too optimistic. In contrast, 2'550 websites sent emails where we did not observe a successful registration. Both observations imply that the validation of a successful registration is error-prone and should be improved.



**Figure 5.2:** Number of received servicing- and marketing emails.

# Additional observations

---

## 6.1 Bot protection mechanisms

During the crawling of test websites, we detected that some web-pages use additional techniques to distinguish a human from a bot. The two most often experienced cases are described below.

### 6.1.1 Sign-up form in an iframe

One simple way of making the registration form inaccessible or hard to detect for an automated process is to include it by using an iframe with the sign-up form embedded. The bot can detect the iframe on the site but since google ads tracking and other services are likely to be implemented using iframes, it is a hard task to classify the correct iframe as a registration form holder.

### 6.1.2 Multistep submit process

Another way to design a registration process which is challenging for automated processes whilst keeping the requirements for humans low is to divide the sign-up form into multiple steps, showing the input fields and the submit button one-by-one. Annotation statistics, made prior to this thesis showed that 8% of websites use this interactive approach, while 4% spawn their registration forms within a JavaScript-driven pop-up.

This could be handled easily by also filling values in all hidden input fields or by designing the automated registration process more adaptive. The first idea could trigger other implemented bot protection systems, since normal users are not able to fill in inputs in the user interface which are hidden.

## 6.2 CAPTCHA solving service secret keys

During the development of the adaptive Python-based CAPTCHA solving process, research for existing solutions which used the same CAPTCHA solving service was done on GitHub and similar public source repositories. Although there were many projects which integrate the AZCaptcha or the imagetyperz service, none of them were designed to adapt to a random page on the internet. Almost all codes were designed to brute-force a predefined login- or newsletter-form with hardcoded parameters in the script.

Amazingly, most of these attack scripts contained the secret keys for the solving service accounts inline as a string variable. Using the `account.balance()` function, we discovered that some secret keys were connected to accounts having paid solving-credits, a few even worth multiple hundred US dollars. These found keys were not used for crawling by our project and the only routine which was executed was the `account.balance()` function that does not deduct the credit-balance of the owner.

## 6.3 Comparison: BuiltWith vs. own usage stats

Our initial decision of which CAPTCHA-system should be automatically solved in the registration process was based on the statistics from BuiltWith, an online service that scans the internet for indications of specific service implementations. By crawling the one million most popular websites and having a rather precise detection algorithm for CAPTCHA implementation chosen to be solved, a reasonable comparisons with BuiltWith and our own results can be done.

CAPTCHAs \ Sources	BuiltWith	Enfbots
reCAPTCHA v2	6.0%	1.06% (+ 0.08%)
reCAPTCHA v3	3.5%	0.48%
hCaptcha	0.038%	0.006%

**Table 6.1:** Distributions of CAPTCHA-systems. The additional number for reCAPTCHA v2 represents the detected invisible versions, as BuiltWith does not differentiate these.

These deviations, which can be seen in Table 6.1, can have several reasons as BuiltWith is based on their own domain list and this thesis is based on the Tranco domain list. In addition, due to only scanning a registration form for any CAPTCHAs, we completely ignore the rest of the website. In contrast, BuiltWith searches the whole webpage. Nevertheless, a similar trend in the distribution of the different CAPTCHA systems can be observed in both data sets.

## 6.4 File system out of space

Initially, the Docker host was equipped with a virtual hard drive of 400 GB storage capacity. Due to the increasing backup time of larger disks, the storage area was kept small. After three weeks of crawling, there was no space left on the device and the Docker containers stopped working. To mitigate this issue, we resized the hard disk to 1.2 TB. By trying to use the gParted<sup>1</sup> tool to enlarge the volumes as well as to resize the filesystem, the startx server had to be used. Since there was no space left, starting startx was not possible anymore. Thus, programs taking up space had to be removed first to free up space.

It would have been much easier to fix this issue if a storage placeholder would have been created from start. This file could have been deleted safely in this scenario. At this point, using the command below, a placeholder of size 500 MB was created.

```
1 $ truncate -s 500M dummy_placeholder
```

## 6.5 Crawling monitoring

Without a monitoring service which would have tracked the progress of the crawling process, the database had to be checked manually every few days if the crawling is still on-going. In the absence of a proper e-mail alerting service, a total of 4 days of computation-time was lost due to multiple late-detected crashes of the crawling process.

---

<sup>1</sup><https://gparted.org/>

# Related work

---

Prior to developing the automated registration process for the enfbots, related work was consulted for out-of-the-box approaches or other useful insights which could be used during the implementation.

## 7.1 OpenWPM platform

Due to the lack of a generic modular platform to perform web privacy measurements (WPM) in easy ways, Engelhardt et al. [3] designed OpenWPM. It contains multiple modules which guarantee a stable operation of the Selenium browser driver, which is integrated underneath this platform.

OpenWPM provides a way for bots to perform measurements on websites in combination with user sessions where the login process is implemented by simply relying on Facebook's federated login system<sup>1</sup>. This is an identification provider which is implemented on a notable percentage of websites. Since the enfbots rely on measurements taken during the registration process itself and do not want to limit themselves to websites supporting these third-party identification providers, the sign-up procedure would need to be implemented by talking directly to the browser driver API underneath OpenWPM. The initial position would be similar to using the standard Selenium automation while minimising the overhead of OpenWPM. Therefore, we decided not to use the OpenWPM platform.

## 7.2 Automated registration

As we did with the enfbots, the paper from Chatzimpyrros et al. [2] describes a generic approach of automated registering to websites and they achieved some impressive results according to presented statistics. In contrast to the

---

<sup>1</sup><https://developers.facebook.com/docs/facebook-login/overview>

enfbots, the validation of a registration may be less sophisticated since it is not described at all. This can be the reason for a high registration success rate of 26.4% in the Alexa top 200'000 domain list.

While running a really good user-tracking detection system, the most impressive test the researchers performed was to inject email addresses into forms without submitting them and leaving the registration process incomplete. After checking the inbox, they received more than 700 reminder emails containing offers as well as discounts. Since we received 2'550 emails not related to a successful registration as mentioned in Section 5.4, they could originate from a similar phenomenon.

## 7.3 Automated sign-in

Jonker et al. [6] used crowd-sourced credentials to automatically login to websites with their automated sign-in bot Shepherd. The goal was to perform scans and enable security testing from within the logged-in area. It is a similar approach to the enfbots but concentrates on the login rather than the sign-up. However, the used crawling process is comparable to our used techniques.

As with the enfbots, keywords are used for the location of the sign-in page and assessed for English as well as other detected languages on a website. In addition to failure keywords for registration validation, the page is scanned for a logout button. From this particular paper, we learned how to speed up the scanning for keywords in link HTML-tags using direct JavaScript instead of the Selenium API.

---

# Discussion

---

### 8.1 Future work

Due to the long testing and crawling process, we were unable to implement new features within the time constraint of this thesis. The following ideas sketch the most promising next steps.

**Machine learning based page classification:** Many websites require as the first step of a registration the choice of an account type or the consent to the terms & conditions. We plan to distinguish the final registration form from these preliminary pages by using a machine learning model. In addition, we can use another model to classify successful registrations. This would allow us to expand the state automaton to also be able to fill in multi-step sign-up forms.

**Keyword detection with ranking:** The current keyword detection can match different types of links or form inputs by matching keywords from the list of synonyms in multiple languages. Currently, the enfbots choose the last matched type, but we plan to compare the likelihood of different matches by ranking them by probability.

**Mitigation of bot detection:** To further increase the number of registration successes, we will migrate the complete codebase to OpenWPM in order to use their sophisticated bot-protection bypassing algorithms, which will cover up the enfbots better. We will accept a possible overhead using OpenWPM and continue talking directly to the browser driver API.

**Solving image CAPTCHAs:** By implementing an additional component, we will complement the CAPTCHA solver with the ability to detect and solve generic image-based CAPTCHAs. The logic for this procedure has already been worked out.



**Adaptive sleeping timers:** Instead of relying on hardcoded sleeping times, these could be managed dynamically based on website speed and other indications. Thus, JavaScript-driven registration form instantiations would be missed less likely.

Hopefully, one day, we will be able to register to almost all websites containing some sort of sign-up process.

## 8.2 Conclusion

The task of monitoring the handling of personal information for marketing purposes is an essential part of enforcing the strict GDPR regulations. Due to the tedious process of having to register on every single website, the only efficient way to achieve this is by using automated web crawlers as explained in this thesis.

The automation arises multiple challenges, which we all addressed in the enfbots implementation. The data gathering of scanning the one million most popular domains provided us further important insights into CAPTCHA implementations, registration process patterns, and we also observed how we need to further improve our crawler by numerous optimisations.

The work presented here can form the foundation for further research that focuses on the topic of web data protection. We are looking forward to further extend the capabilities of the crawler and gather even more insights into the data handling procedures of online services.

---

## Bibliography

---

- [1] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. Web runner 2049: Evaluating third-party anti-bot services. pages 135–159, 2020.
- [2] Manolis Chatzimpirros, Konstantinos Solomos, and Sotiris Ioannidis. You shall not register! Detecting privacy leaks across registration forms. In *Computer Security*, pages 91–104. Springer, 2019.
- [3] Steven Englehardt, Chris Eubank, Peter Zimmerman, Dillon Reisman, and Arvind Narayanan. OpenWPM: An automated platform for web privacy measurement. *Manuscript*. March, 2015.
- [4] European Parliament, Council of the European Union. General data protection regulation (GDPR): Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec.
- [5] European Parliament, Council of the European Union. Directive 2002/58/EC of the European Parliament and of the Council of 12 July 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector (directive on privacy and electronic communications), 2002.
- [6] Hugo Jonker, Stefan Karsch, Benjamin Krumnow, and Marc Slegers. Shepherd: A generic approach to automating website login? 2020.
- [7] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. February 2019.

- [8] Amit M Potdar, DG Narayan, Shivaraj Kengond, and Mohammed Moin Mulla. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428, 2020.
- [9] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. CAPTCHA: Using hard AI problems for security. pages 294–311, 2003.
- [10] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollén, and Martin Lopatka. The representativeness of automated web crawls as a surrogate for human browsing. pages 167–178, 2020.

---

# Appendix

---

## Sources

We decided to publish the complete source code including all the results to allow others to reproduce our study and further improve the described and used techniques in this thesis. The source code can be found on GitHub<sup>1</sup> while the crawling output is located on Google Drive<sup>2</sup>. To simplify the inspection of the crawling output files, the files were compressed into a total of 1'365 zip archives. The name of each zip file indexes the first two characters of all analysed domains that are contained in this archive. Please note that all credentials used for user registration on websites as well as database connection credentials were removed prior to uploading.

---

<sup>1</sup><https://github.com/PatriceKast/enfbots/>

<sup>2</sup><https://drive.google.com/drive/folders/1EA1RQwcUiKayJufaZs-BVQcxpSxQjZc2>



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Enforcement Bots: Nothing can block us!  
Automating website registration for GDPR compliance analysis

**Authored by** (in block letters):

*For papers written by groups the names of all authors are required.*

**Name(s):**

Kast

**First name(s):**

Patrice Michael

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

**Place, date**

Wetzikon ZH, 01.03.2021

**Signature(s)**

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*