Designing a generic web forms crawler to enable legal compliance analysis of authentication sections

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Designing a generic web forms crawler to enable legal compliance analysis of authentication sections

Master Thesis

L. Lodrant

January 17, 2022

Advisors: Prof. David Basin, Karel Kubicek
Department of Computer Science, ETH Zürich

Abstract

While users deserve security and privacy when using web services, these properties are at odds with the financial interests of website owners both in terms of work required to keep websites secure and revenues generated by exploiting sensitive data resulting in a violation of the user's privacy. Countries, therefore, introduced regulations to balance the inequity. Namely, European Union's General Data Protection Regulation (GDPR) specifies that any data collection and processing can only be done with the informed and specific consent of the user, including sharing of the said data with 3rd parties. Automated and large-scale detection of violations and security flaws is difficult because of the non-standardized behavior of website authentication mechanisms.

We developed a web crawler for detecting and submitting mainly registration web forms. This crawler enables novel privacy and security research on a larger scale than was previously possible. The completely automated crawler can navigate the site to find the required form, fill the form, avoid bot detection mechanisms, submit the form, and validate the submission success. In 17 days, we crawled over 600,000 domains intending to create new user accounts. Our automated crawler detected a sign-up form on 22% of all the reachable websites with a 6.4% registration success rate. We have also received at least one email from 2.3% of all crawled pages. This significantly surpasses the prior version of this project and the best widely-used published tool.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Karel Kubicek for his continuous guidance, feedback, and for keeping me motivated throughout the process. Without you, this thesis would never have been possible.

I am also deeply grateful to Prof. David Basin and the Information Security Group for providing me with the resources required to conclude my research.

Additionally, I would like to thank Patrice Kast for joining the weekly meetings and providing his insights.

Finally, I would like to extend my thanks to KastGroup GmbH for providing the computational resources required to evaluate my work.

Contents

Chapter 1

Introduction

In recent years much of daily life has moved to the internet, the transition only spurred by the recent pandemic. Unintentional bad practices, non-ethical behavior, and malevolent actors pollute the online landscape and infringe on the privacy and security of all users.

E-commerce websites are known to either intentionally or unintentionally share their email marketing lists with 3rd parties [7] resulting in the delivery of unsolicited emails. Similar practices are employed by the ever-popular free web services, whose financial model relies on ad revenue, where the user's private data is the means of payment for the services rendered [17]. European General Data Protection Regulation (*GDPR*) [10] has been a big step forward in providing a legal framework for the protection of the user's privacy on the internet. It requires freely given, specific, informed, and unambiguous consent before any collected data can be processed and narrows the scope in which the data can be used. The technological means to mandate privacy is then defined in ePrivacy Directive (*ePR*) [9], which, for example, forbids sending unsolicited marketing emails and sharing the user's email addresses with 3rd parties unless the user consents to the practices. According to Kubicek et al. [12], 21.9% of websites contain potential violations of these rules.

Authentication mechanisms can also contain various vulnerabilities, which can lead to serious information leaks. Drakonis et al. [6] have audited more than 25,000 domains and have detected almost 10,000 domains where an attacker could access the user's private data or hijack the authenticated session. Calzavara et al. [1] have shown that even the most popular pages on the internet are not exempt from such vulnerabilities.

The website registration process itself is potentially susceptible to the leakage of personally identifiable information too. According to Chatzimpyrros et al. [2], around 5% of webpages intentionally or unintentionally leak users'

data to 3rd-party trackers without consent. Starov et al. [19] have shown that the same holds for contact forms as well.

The study of all the presented topics on a large scale is difficult because of the lack of tooling required for automatic interaction with authentication mechanisms. Selenium [4] and Pupeteer [3] provide only low-level access to web browser automation, while the OpenWPM [8] framework mainly supports passive web privacy studies.

The goal of this thesis was to improve on the work done by Kast [11], who developed a crawler for automated website form interaction meant to facilitate GDPR compliance analysis. The main focus of the crawler remains to automate the registration of users to websites, which will in the future facilitate research on the usage of user-provided private data by websites. Most of the challenges that we dealt with in this thesis come from the fact that website interaction flows are not standardized, and the developers have complete freedom to design it however they want to. There are many common patterns, but in the end, almost every website has a unique implementation, which the automated crawler has to account for.

The past version of the crawler followed a very rigid scheme. In simplified terms, the crawler used keyword-matching to detect the registration and login page, assumed the form with the most password fields is the registration form, fill in the form, solved the CAPTCHA, and submitted the form. This works for a lot of websites, there are also many more complex websites where this simple scheme can fail at. Notably, any website with a shared page for registration and login, multi-stage forms...

We have reworked the crawler's website navigation module, which now uses keyword-matching of links to fill a priority queue. The pages in the priority queue are ordered by how likely they are to contain the desired content. That, combined with an improved form classification algorithm, makes the updated crawler much more adaptable and, therefore, more likely to detect a registration form. It also makes it much easier to change the end goal of the process (e.g., to detect newsletter subscription forms). We also improved the form detection, filling, and submission. Lastly, we have improved the overall reliability of other parts of the crawler and prepared a test suite that covers most of the cases that can happen during a real-world registration scenario.

Overall we have noticed a significant improvement both in terms of successfully detected registration forms and overall registration success rate. In the evaluation crawl of the Tranco [14] 1M most popular websites, we successfully detected a registration form on 20% of all successfully loaded websites and have estimated the submission to be successful on 6.4% of them. By reviewing the emails received to the automatically generated email addresses, we observed that 2.3% of all the loaded websites sent us at least one email, producing a guaranteed lower bound on the overall success rate.

Chapter 2

Initial crawler state

The base version of *EnfBot* was first introduced by Kubicek et al. [12]. That version still delegated parts of the registration process to the user. It was written in Python and used Selenium to automate a headless Firefox browser. Later it was significantly improved by Kast [11], making it completely automated. He also introduced a distributed crawling platform to crawl the most visited 1M websites collected by Tranco [14]. This resulted in a successful registration on approximately 40 000, i.e., 4% of the 1M crawled pages. At the end of my thesis, we aimed to repeat this crawl to compare the versions between themselves.

The basic operating procedure that was used at the time is represented in the Fig. 2.1. The main goals of the crawler were, given a website, to register on it if possible, and to find its privacy policy and terms of use.

The crawler can be divided into two almost independent parts, the navigation, which is responsible for crawling through the website, and the registration module, which is in charge of filling and submitting the form. Here we will analyze the shortcomings of both parts. Navigation was based on matching the text present in the links to keywords that were manually assigned to desired categories. It supported 39 languages (most of the official languages of the European Union), but German and especially English were the most tested and reliable. The registration module used a similar keyword-based approach to classify input fields and fill them with appropriate data.

2.1 Limitations

The described crawling flow has several limitations in the navigation and registration modules. Some were already presented by Kast [11] while others were noticed during work on this thesis.

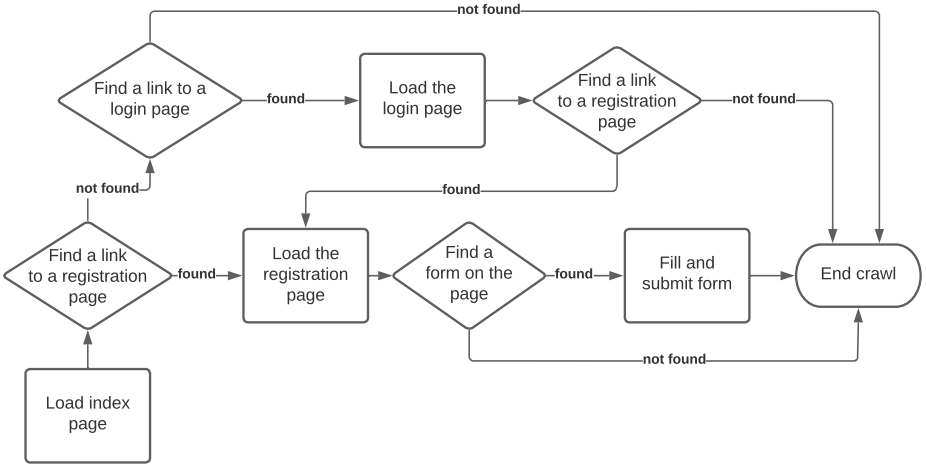


Figure 2.1: A representation of the crawling flow utilized by the previous version of the crawler.

2.1.1 Navigation

The previous crawler implementation depended on a specific registration structure depicted in Fig. 2.1, the rigidity of which prevented it from successfully navigating through the more complex websites.

The link keyword matching algorithm failed to consider links that matched multiple page types. Instead of determining the best match, it selected the first type detected.

This crawling scheme also only attempted to load the first link to a registration page it encountered. It failed as soon as that link led to a page that did not contain a registration form, even though the second-best match might be the correct one. Examples of this are login pages; they share keywords with registration pages (e.g., account), so one is often mistaken for the other. Therefore, the crawler should be adaptable and try to find registration pages on all pages that match at least some of the related keywords. The index page sometimes contains a registration form, e.g., on members-only portals.

The crawler also had problems with websites where login, registration, and possibly some other forms are present on the same page. The algorithm used to determine which form to submit often resulted in false positives, thus submitting login, newsletter, or contact forms. While this does not affect the actual success rate, it leads to misleading results.

Many other problems occurred only on a small subset of pages but together accounted for a significant amount of pages that the previous version of the crawler was not able to successfully process:

- index landing pages contain a single link/button requiring interaction

(e.g., Continue to site),

- websites require age confirmation,
- websites contain irrelevant links that lead off the site (e.g., youtube.com/user/partnersupport) matches the keywords for login page), and
- websites require navigating by more than one step between the index and registration pages.

2.1.2 Registration process

Once a form had been identified, the crawler first classified all input fields. An input field was often matched to the wrong label, which led to misclassifications. There was no way to distinguish between different select boxes, meaning an input field for the birth date was often filled out wrongly. This could lead to failed submission in cases where age was validated.

The crawler also had issues detecting certain types of CAPTCHAs and did not correctly solve them again if the first submission attempt was unsuccessful. For example, the way reCAPTCHA worked relied heavily on a particular way of its integration into the website. The crawler was therefore unable to solve reCAPTCHA properly on websites with a more complex setup or middleware libraries.

The most problematic part of the form interaction was the registration success detection. There were several steps in this process, each with its own flaws that made the results of any crawl unreliable:

1. It analyzed the forms present on the page using the same flawed algorithm is used to detect the form in the first place. If it detected a form, it decided that the submission was not successful. Because a login form was always detected as a registration form, this meant that every site that redirected you to a login page after successful registration was marked as unsuccessful. At the same time, any multi-step registration form that did not have an email field on the second form was detected as successful.
2. It treated redirects as successful registrations. Not necessarily correct, e.g., redirect to an error page
3. It checked for success and failure keywords present on the webpage. The keywords used were extremely generic. Therefore this often led to a false positive in cases where any of the keywords were present on the page after submission.
4. In case nothing else was detected, it assumed that the submission was successful. This proved to be a very optimistic and unreliable behavior.

Chapter 3

Crawler rework

The core of the improved EnfBot crawler is a page navigation algorithm instrumented by a page-state automaton, which replaces the previous sequential webpage navigation implementation. We model the registration process as a set of states and a set of operations that can determine the current state we are in. The crawler can then use this information to determine the best course of action in order to achieve the three main goals: register on the web portal, find the terms of service, and find the privacy policy. The crawling process can be roughly split into two independent parts, finding the required pages and performing the actual registration.

Fig. 3.1 visualizes the navigational part of the website crawl. It only requires the URL of the index page to be put into the navigation queue and then works autonomously until the crawl goals are met, or until it reaches the maximum number of subpages, it is allowed to crawl.

When the desired type of form is detected we enter the registration phase. In Fig. 3.2 we represented the steps the crawler takes when registering on a website with an unknown structure. A separate module is in charge of every step of the process; starting with input classification, then filling in the forms, submitting the form, and finally checking the status of the current submission.

This modularization of the process allowed us to develop and improve each step in isolation and, in a similar vein, present each of them separately in the following sections.

3.1 Keyword matching

One of the key parts of the entire crawler is the keyword matching algorithm. We use it to classify links, forms, input fields, and even to check the submission success. The algorithm takes a string and a set of categories, each with

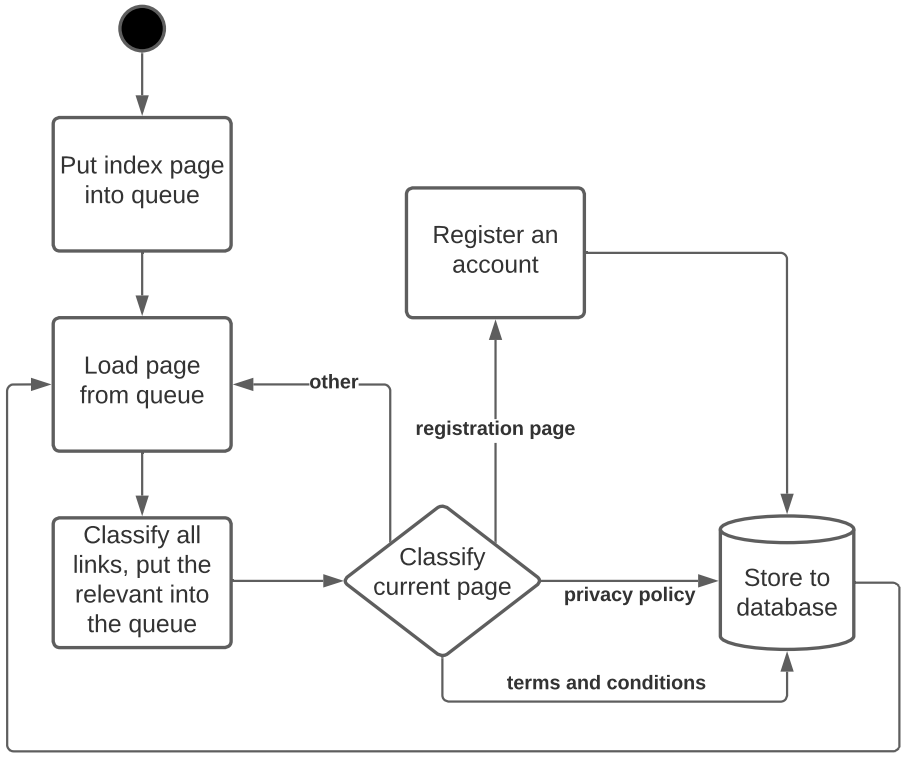


Figure 3.1: A visualization of the way the crawler navigates through a single website.

its own keywords. The task is to select the category whose keywords match the given string best.

Let's illustrate the functionality with an example. We would like to know what kind of data a given input field with an associated label should contain. For simplicity, let's assume the only options are *email*, *full name* or *username* (in practice the number of possible categories for input fields is much larger).

The label text is "Your username" and the considered input field categories and their corresponding keywords are:

- email: email, e mail, mail
- full name: full name, name
- username: username, userid, user, usr, nickname, login.

We know from the label that the expected output is *username* input type. Unfortunately it contains the string *name* which also matches the type *full name*.

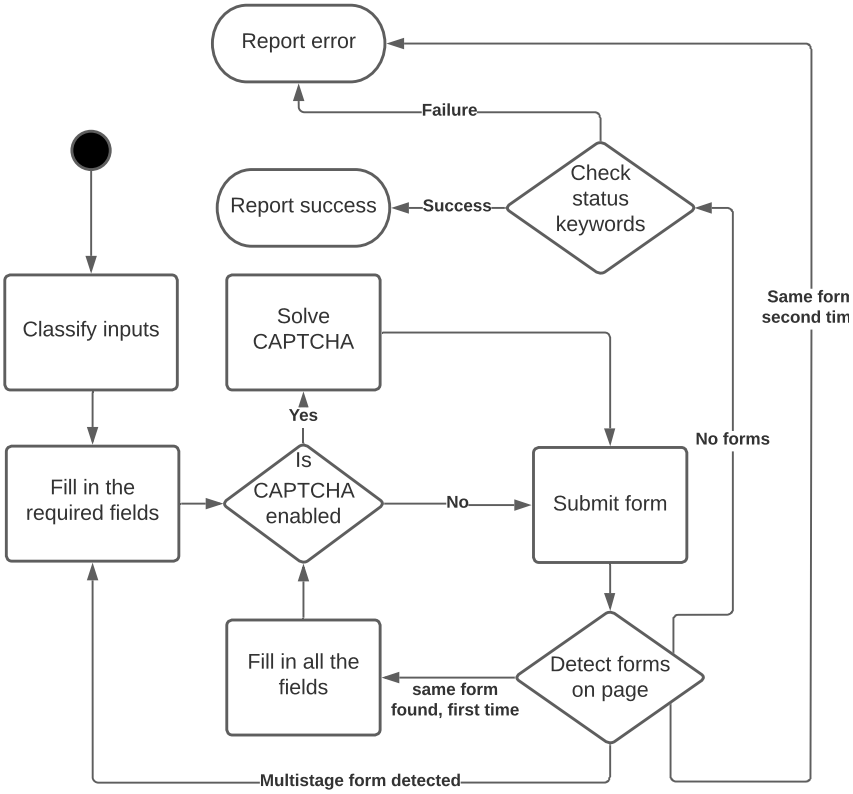


Figure 3.2: A representation of the registration procedure the crawler follows on each detected form.

Our keywords-matching algorithm assigns a score to each type depending on how well the type matches the string. Then it compares the scores of all types and outputs the one with the highest score.

The string is first split into separate words by splitting on all whitespace characters, special characters, and all case changes. We will call these *label* words.

We calculate the matching score for each category separately. First, we compare the list of label words and the list of keywords pertaining to the given category and look for matches. For each match, the score for the type is increased by $1/n$ where n is the position of the matched keyword in its list. In our example, this happens in the type *username*, where the first keyword in the list is matched completely, meaning the resulting score is 1. Note that for keyword phrases (like "full name"), all the words in the phrase have to be matched in order.

If no full word matches are found, we check whether any keyword matches

just a part of a label word or vice versa. In our example, this happens with the word "name" for the type *full name*. In this case, the score is increased by just $1/(n+5)$ making the increase smaller for partial matches. In our example, this would give the type "full name" the score of only 0.15, thus classifying the entire input as a username field.

3.2 Navigation

The navigation module's task is to scan the website for links and other clickable elements and select the next page to visit in order to achieve the current goals. The navigation is generic, and the crawling goals can be adapted easily. E.g., the crawler can be extended to prioritize logging in and navigating through the pages only available to the authenticated users, signing up for newsletters forms instead of registration, only looking for the privacy policy....

3.2.1 Hyperlink classification

Each time a new website is loaded, all the links present on the page are collected together with any accompanying text. The text from the link's label, title, destination address, and accessibility attributes is then matched to predetermined lists of keywords using the keyword matching algorithm described in Section 3.1. The following types all have assigned keywords for each of the supported languages:

- registration,
- login,
- terms and condition, and
- privacy policy.

The result of the keyword matching algorithm is the best matching link type with its assigned score. In case no matches were found (the majority of the time), the URL was classified as *other*. It is important not to discard non-matched hyperlinks completely because they allow us to enter the site from a landing page where we would otherwise be stuck.

3.2.2 Navigation queue

The hyperlink classification assigns URLs their corresponding category. These are put into a priority queue which prioritizes the hyperlinks that match the goals that we still have to achieve during a particular crawl. Given the goal of the crawler, the links leading to a registration page are the top priority. Second are links leading to a login page, as these often contain registration

forms or at least links towards a registration page. Third are links to the privacy policy and terms, which are also important for the legal analysis, but they are only collected after registration. All the other links share the lowest priority.

Links within a category are ordered according to the keyword matching score regardless of the time of insertion into the queue. There is also a limit to the number of visited links allowed in each category. This prevents the crawler from getting stuck on a particular website where there are a lot of incorrectly classified links. This happens when the entire portal resides on a subpath containing one of the keywords (e.g., `register.com`).

3.3 Page classification

The hyperlink classification only orders the navigation queue, which determines the next crawled page, but it does not decide about actions to be performed on a particular page. The text in the link contains too little information and thus can be inaccurate. Therefore the whole content of a loaded page needs to be considered in order to decide which action to perform next. This includes determining the general type of the page and detecting specific page components (e.g., forms) of interest. In the following sections, we discuss the possible design options, our decisions between them, and finally, the limitations and proposed future additions for this module to be feature complete.

3.3.1 Full page classification

The initial idea was to develop a machine learning model to classify each loaded page. The model would distinguish between registration pages, login pages, privacy policy, and pages containing the terms and conditions. With this model, we could still use the existing keyword-based navigation, and in addition, the model would confirm that the loaded page matches the type detected in the link.

We collected a dataset of around 1000 webpages in English using the results of the first crawl [11]. During the manual labeling process, it became evident that all the login and registration pages can usually be quite easily distinguished from all the others by the presence of login and registration forms, respectively. This means that the classification task can be separated into two sequential tasks, first distinguishing between form and non-form pages and then subclassifying each group. Since registration is the primary focus of this crawler, the biggest emphasis was put on the form group, which we will cover here. The subclassification of the second group containing privacy policy, terms of use, and all the pages that use similar keywords (e.g., articles about privacy) remains as future work.

During the labeling procedure, we also observed that many pages were hard to classify into any of the chosen groups since they contained multiple forms of different types. For example, a webpage can contain a login form on the left, a sign-up on the right, and a newsletter subscription form in the footer. Initially, we classified these pages into a special "MULTIPLE" category, but that only helps for the crawler navigation, leaving the decision about which form to fill and submit unsolved. It was concluded that the initial idea of page classification for this task is cumbersome and that each form found on the page needs to be classified separately.

3.3.2 ML-based form classification

By focusing the classification on forms we eased the labeling process because forms, unlike webpages must have a singular purpose. All the pages from the first dataset collection were crawled again and checked for the existence of forms. We then collected the screenshots and source code, which we labeled manually. We developed a generic annotation interface to simplify the labeling process, see (Appendix A.1). The form was pre-classified by a simple heuristic that takes into account the number of specific types of input fields and the existence of certain common keywords. The final annotation was then decided by the author. The dataset contained 426 forms, of which there were 12 contact forms, 32 login forms, 139 newsletter sign-up forms, 163 registration forms, and 80 others. Since it collected all the forms found on a webpage (not just the first), it should be representative of the forms the crawler is going to encounter.

The most obvious features that can be used to classify a form are the types of input fields present in it. The existing input field classification algorithm only worked on forms loaded in the web browser. Loading each form in the browser just to extract features from it would add a big overhead to the model and make it much less portable. To create a proof-of-concept model, we just used the HTML type attribute to classify the fields into categories email, password, text, and select. The number of elements of each type was used as a feature, as was the total number of all non-hidden inputs. Additionally, we trained a bag-of-words model on all the text present in the forms and the URLs from the dataset and selected the 100 most common ones to use as predictors.

Using all these features, we trained an *XGBoost* model, which reached an accuracy of around 71%. This was expected because of the limited set of features used and the limited amount of data it was able to train on. To improve the accuracy, we needed to remove the input classification algorithm's dependency on Selenium in order to be able to use it for the purposes of feature extraction.

Feature importance analysis also showed that due to the wide variety in the detected keywords, the bag-of-words model did not have enough training data. Collecting a bigger and more nuanced dataset would require a big web crawl, especially if we wanted separate models for all the supported languages, so further research into this area was postponed.

3.3.3 Traditional form classification

Instead of training a model on an insufficient amount of data, we developed a more detailed deterministic algorithm for form classification. It primarily depends on the number of important input fields and falls back to keyword matching in cases where that is not enough to classify the form with the required accuracy.

A simplified scheme is visualized in Fig. 3.3. The main branching point is the number of password fields. Almost every form with more than one password field is a registration form, the exception being password change forms which we do not expect to encounter because we are not logged in at the time of the registration. Forms with a single password field are a bit more divisive because they can also be used for login. In this case, we assume that login forms usually do not contain many personal fields (e.g., name, country, phone number, birth date ...). Even forms without any password fields can be used for registration (multi-step, generated password ...), although they are more often used for newsletter subscription or contact. We assume that personal fields are not common in newsletter sign-up forms either. The decision between contact and registration form is a bit harder. For it to be considered a registration form, we require the submit button to contain a keyword linked specifically to registration (e.g., sign-up, register ...) which is rare in contact forms.

Using the traditional algorithm on the same test set we used for the evaluation of the ML model, we observed accuracy of 82%, which is significantly higher than the 71% achieved by the ML model. In Fig. 3.4 you can observe the confusion matrix of both developed models. The test dataset is too small to draw any serious conclusions, but you can observe some trends. The ML model tends to disregard login forms almost completely, while the traditional model often mistakes actual registration forms for login forms. We also noticed that there is higher false negatives are more common regarding registration forms than false positives. Manual review has shown that the misclassified login forms are in large part the ones with just an email and a password field, where keyword-matching is not enough to distinguish them apart.

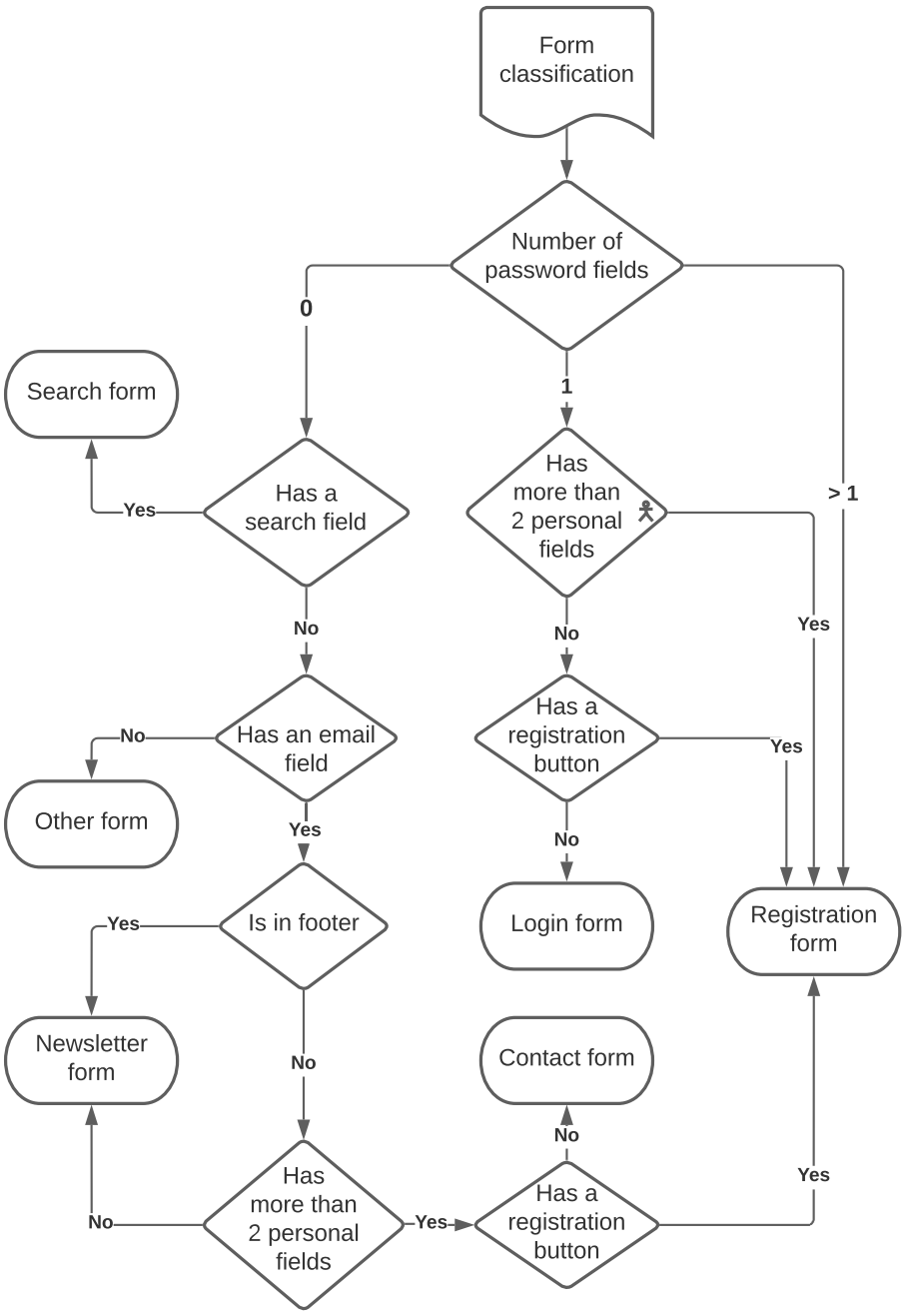
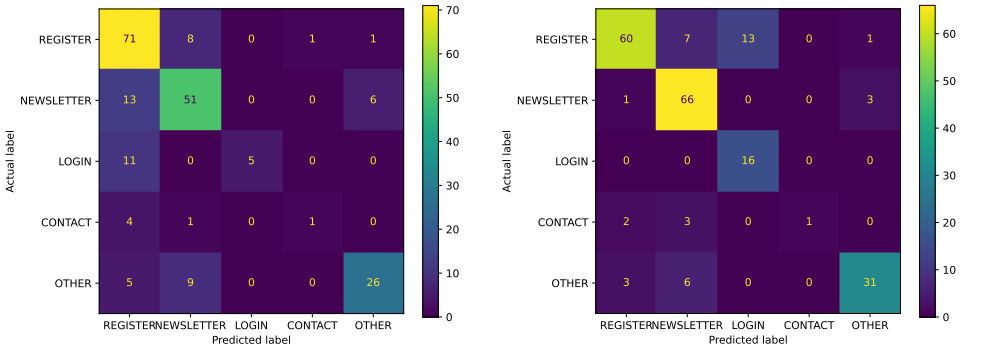


Figure 3.3: Traditional form classification scheme



(a) Confusion matrix of the ML model. **(b)** Confusion matrix of the traditional algorithm.

Figure 3.4: Comparison of ML and traditional classifiers.

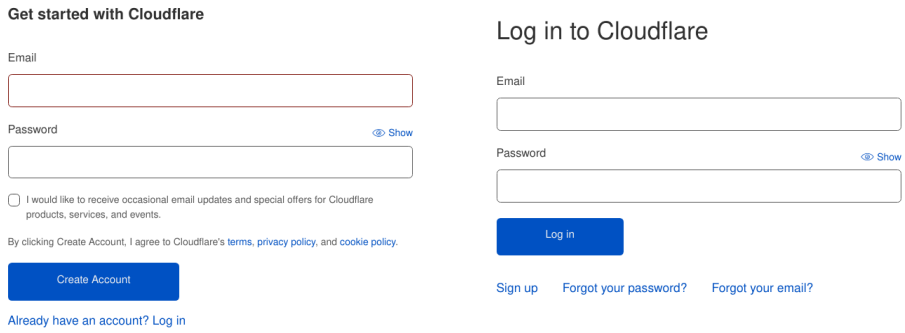
3.3.4 Limitations and future work

The deterministic algorithm still has problems differentiating between certain groups of forms, namely:

- newsletter sign-up forms, contact forms, and multi-step registration forms
- login forms and registration forms with only an email and password field

The type and number of typed input fields alone are insufficient to distinguish between the forms in Fig. 3.5. To distinguish these forms, we would need to use more sophisticated NLP procedures or new features such as in-page location. Simply using more training data would possibly improve the model as well. Nevertheless, the deterministic form classification is not the main limitation of the system, as we discuss in Chapter 6, so we did not explore the ML classification of forms further.

The form classification differentiates only between registration, login, and all other pages. We still need a model that differentiates between a page with the terms of use and privacy policy from other pages. These pages are usually monotonous and long as they contain the entire legal document. Hence we can use features such as text length, number of images, and links, together with text vectorization, fed to NLP models. We can use the data from the final crawl, but we would still have to label or weakly supervise the classification model, similarly to the one studied by Mekala and Shang [16]. This is also left for future work.



(a) sign-up form

(b) login form

Figure 3.5: An example of forms with the same input fields but different meanings on the same site.

3.4 Filling the forms

Once a registration or newsletter subscription form is detected, we must fill out all its input fields. The main objective is successful submission; the accuracy of the data submitted is just a secondary concern. Each run of the crawler uses a provided persona to fill in all the personal fields, while the email address and the username are generated for each execution separately to allow the creation of multiple accounts on the same page. We leave any field that is not matched to a known category empty in the first submission attempt. If the submission attempt is not successful, we resort to filling in all the fields randomly, i.e., with a random string in case of text boxes and a random option in case of select menus and radio groups. The randomly generated strings are made sure to adhere to any known restrictions like max length, min length, and regex pattern.

3.4.1 Input field classification

Before filling in an input field, we first need to determine the type of data each input field is supposed to contain. Additionally, we also need to detect any special formatting requirements for certain input types (e.g., dates, phone numbers).

There are three main sources from which we can infer the type.

1. HTML `autocomplete` attribute.
2. HTML `type` attribute.
3. Strings: label, placeholder, value.

The most reliable way to classify fields is the `autocomplete` attribute given to any input field. It is a part of HTML5 standard specification[1] and its main goal is to help the browsers and password managers automatically fill the forms in the name of the user. Its value directly determines the type of input field. For example, the value "username" would tell us that the forms expect this input to contain a username. Unfortunately, only a small proportion of webpages – 18% by the estimation from our evaluatory crawl - actually add the attribute to the input fields and forms.

If the `autocomplete` attribute is not present, we use the HTML `type` attribute to narrow the range of possible input types (e.g., a checkbox cannot contain an email address) and then perform the same keyword matching algorithm already described in Section 3.1 to classify each field. The keyword matching is performed on the placeholder, value, class names, and the label, which all often contain keywords related to the type of the input field.

3.4.2 Inserting values

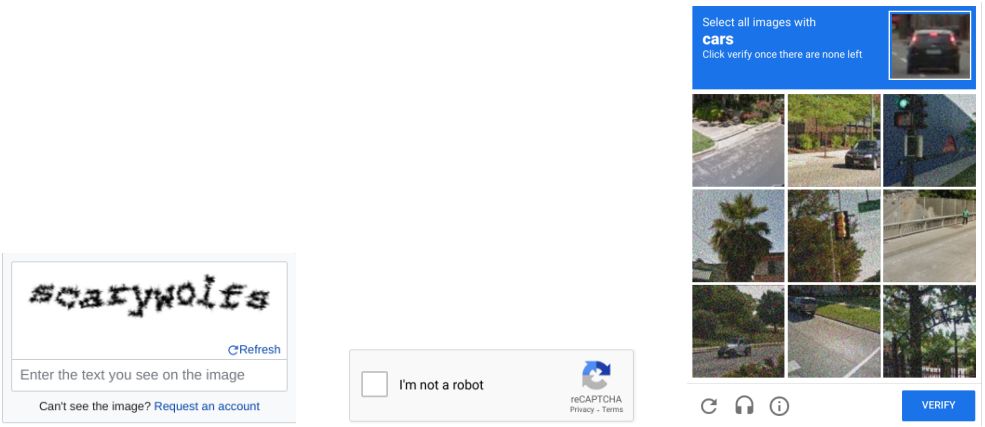
Most input fields the crawler encounters are just text boxes. In this case, we insert the value by simulating the keypresses using Selenium instead of just directly filling in the value using JavaScript. This is done to make sure that all the event listeners on input fields are triggered. Without this, some forms might not be submittable since the input value verification was not activated yet.

Select boxes and radio groups are handled similarly. In both cases, we extract the label and value assigned to a particular option and use our keyword matching algorithm to compare it to the value we would like to fill in. The option with the best score is then selected using Selenium's built-in functions.

3.5 CAPTCHA solving

Many websites have measures in place which prevent bots from registering in order to prevent malicious actors from creating new accounts. Without it, they would be susceptible to spambots, DDoS, and other attacks. For our crawler to work, we require a way of circumventing these protections. Commonly, a website presents us with a challenge that is easy for a human to solve but hard for a bot to solve automatically. Systems that provide such functionality called CAPTCHA (for Completely Automated Public Turing Test To Tell Computers and Humans Apart) have been in use since 1997. The term was coined later by von Ahn [20].

[1] https://html.spec.whatwg.org/multipage/form-control-infrastructure.html-#attr-fe-autocomplete



(a) image captcha **(b)** reCAPTCHA checkbox **(c)** reCAPTCHA image

Figure 3.6: Different types of CAPTCHA systems you are likely to encounter on a webpage.

We implement detection and processing of multiple CAPTCHA systems but use an external service *2Captcha*[2] to solve them. In most cases, that means we collect all the information required to solve the challenge and send it to their API, where it is solved using a combination of machine learning and human labor. The results are then passed to us and used when submitting the form.

3.5.1 Classical CAPTCHA

Typical CAPTCHA implementation is a variation of a distorted image from which the user is supposed to read characters, a riddle to be solved, or an audio recording to be typed out. An example of such CAPTCHA is represented by Fig. 3.6a.

Currently, our crawler only supports image recognition, but in the future other forms could be added in a very similar fashion. The detection of the image and answer field works in the same way as the classification of form input fields. We decide if a certain input field is part of a CAPTCHA by checking for certain keywords, the existence of nearby images, and the existence of IFrames often used by external CAPTCHA providers. After that, the image to be recognized and a screenshot of the surrounding area, which might contain additional instruction, is sent to the external service. If the service solves it successfully, the response is filled into the provided input field, and the form is ready for submission.

3.5.2 reCAPTCHA and hCAPTCHA

The second category of CAPTCHA systems our crawler supports are those provided by reCAPTCHA and hCAPTCHA. As hCAPTCHA is a drop-in

[2] https://2captcha.com/

replacement for reCAPTCHA for those, who would like to migrate away from using the Googles service due to privacy-related or other concerns, we will just refer to reCAPTCHA in the following section but note that everything stated holds true for hCAPTCHA too.

These systems first try to determine if a user is a human without requiring a challenge to be solved. Usually this means just showing a checkbox needing to be checked, or in the case of reCAPTCHAv3, not even that. While the inner functionality of these systems is not known to the public, it has been researched by Sivakorn et al. [18]. They concluded that the widget analyzes the browser environment, tracks the user using cookies, and also analyses the user's interaction with the site in order to determine if the user is human. If the analysis is not able to confirm the user to be human, a challenge is still presented and expected to be solved. In Fig. 3.6 you can see both modes of operation.

An automated solution of reCAPTCHA differs from manual interaction as it does not involve ticking the checkbox or choosing any images. We first extract the parameters that were used to render the reCAPTCHA widget and then send them to the CAPTCHA solving service, where they render the widget on their own and just send us the response required to submit the form.

We capture the parameters and later provide the website with the answer to the CAPTCHA by replacing the official reCAPTCHA JavaScript with our own NOOP implementation. To the website, the JavaScript API is indistinguishable from the original implementation, while in reality, it does not render any CAPTCHA challenges but just forwards the parameters to the solver. For a more detailed explanation of this system, see Appendix A.2.

3.5.3 Limitations

Not all sorts of CAPTCHA systems are supported yet. Our external provider supports solving more types that we are currently detecting, f.e., text CAPTCHA, GeeTest, CapyPuzzle... From the data collected by BuiltWith[3] we can conclude that supporting these would yield only negligible improvements, but this is still an avenue we could pursue for the sake of completeness.

3.6 Submission validation

After submitting the form, it is important to check if the registration was successful or not. This module is essential, especially in the analysis of the crawler's performance. Errors that arise from the submission validation can relay a wrong impression of how the rest of the crawler is performing. There

[3] https://trends.builtwith.com/widgets/captcha

are many ways a website can respond after the submit button is pressed. We identified the most common success and failure scenarios and developed a validation pipeline to distinguish between them.

Success scenarios

- We are redirected to a success page.
- We are redirected to a login page.
- A success message is rendered on the same page.
- We are taken to the next step in the registration process.

Failure scenarios

- The same form is shown with some error messages.
- We are redirected to an error page.
- No redirect, no form displayed, just an error message in its place.

3.6.1 Form detection

First, we rerun the same form detection algorithm we used to detect and classify registration forms to see if the page still contains some forms after submission (see Section 3.3.3). Each form is compared to the form we just submitted for approximate equality. We are interested in knowing if we were presented with the same form again, but we allow for some differences. Every time a page is loaded, we append a UUID to each input field and form. By comparing the UUID of the old and new forms, we can reliably decide whether the form has been reloaded. Additionally, webpages often add or remove particular elements from the form after a failed submission, e.g., a CAPTCHA field, error fields, or do not ask you about fields that were OK in the first submission. Therefore, we penalize each difference of inputs by 1 point. The same penalty of 1 point is assigned for any difference in the "action" and "name" attributes of the form. In the end, we decide whether two forms are equal based on the aggregated penalty and number of input fields. For forms with fewer than three visible inputs, the penalty must be 0 for forms to be considered equal. For forms with three to five inputs, the penalty must be lower than 2. Furthermore, for forms with more than five inputs, the penalty must be lower than 3.

As we have already mentioned in Section 3.4, in case the same form has been presented again, the crawler will attempt to submit it again. This time we will also fill any field that was unable to be classified and was therefore not filled in during the first attempt, using random data. When that has already been done, we will assume that the registration has failed.

If we detect a registration form that is not equal to the submitted one, we assume that we are working with a multi-stage registration process and continue by filling in and submitting the next form.

The final possible scenario is that no registration forms were detected. In this, we continue with our submission validation pipeline.

3.6.2 Keyword detection

The second step in the detection pipeline again focuses on a set of keywords that can represent a successful or unsuccessful registration. Some of these keywords are extremely general (i.e., success, failure,...), so they tend to produce quite a lot of false positives. To combat this, we run the same keyword matching algorithm before and after submission. The keywords encountered in the first one are not taken into account when checking for the results since they were already present on the page even before submitting.

3.6.3 Redirect detection

Next, we check if we were redirected to a new URL during our registration procedure. This still does not necessarily mean that the registration was successful because we could also be presented with an error page. In our experience, this is rarely the case; a failure usually keeps at least the form or an error message on the same site. As of now, we have not figured out a way to be more precise, but this might require further examination.

3.6.4 Email validation

The last part of the submission pipeline is email verification. This is done independently of the results of the previous detection methods. We wait for up to two minutes to see if our newly generated email address has received any emails. If an email arrives, we conclude that the submission must have been successful. Secondly, we check the email for any link that could be classified as activation and visit it, hopefully activating the account and completing the registration process. While we have noticed websites that require user interaction to activate the account after clicking the link successfully, we currently do not support that interaction.

3.6.5 Future work

If none of our detection methods succeed, the submission status is marked as unknown. The results of the final evaluation show that quite a large amount of pages fall into this category, meaning the success detection module is still far from optimal.

By intercepting the HTTP requests used for form submission, we could intercept the HTTP status code of the response and use it to verify the submission status. While a successful HTTP response does not in any way indicate a successful registration (many failed attempts are just served with status `200 OK` and an error message), an error status should be a good indicator of failure. This is especially true for any AJAX-based forms where the HTTP status is more commonly taken into account.

The keyword detection algorithm could be improved a lot using a variety of NLP techniques. We have already looked into training a BERT [5] model to distinguish between successful and unsuccessful responses. The dataset required for this was collected later, during the evaluatory crawl we will discuss in Chapter 6. There is also the problem of the amount of irrelevant text always present on the page. An interesting approach would be to devise a way of ranking all the text content by the amount of attention a human observer would direct to it. This would allow us to prioritize the detection on the text that is most likely to tell us the result of our submission and ignore things like the footer and the header of the page.

Darkonakis et al. [6] have also proposed that loading the index page again and checking for the presence of any of our personal information can be a good indicator of a successful registration attempt.

Chapter 4

CI/CD pipeline

The previous implementation of the crawler relied only on large-scale crawls for analyzing how well it performs. The approach had multiple problems. It was limited by the website selection, and for different samples, the success rate could differ significantly. Fixing website samples did not help, as with time, the websites would change, and even a tiny change could have an absolute impact on results. Additionally, recrawling a fixed set of sites every time also proved problematic because it would create a new account with our credentials on each site every time. During the work done on this thesis, the pipeline would be run more than 150 times.

Additionally, code improvements to handle a new subset of special cases increase the crawler's complexity. Over time it became hard to follow which parts of the code still function correctly and which parts were broken by the improvements of other modules. A big crawler test run does not help here because an improvement for one class of webpages and regression for another can nullify themselves.

We addressed this by extending the project with a suite of unit and regression tests. All of these are automated using Gitlab CI/CD and test the functionality of the crawler without any interaction with external websites.

4.1 Functional testing application

Some parts of the code like keyword matching and classification module are easily unit-testable, just by providing the crawler with fixed examples. For other functionality, a strictly unit testing approach would require a lot of mocking. Instead, we have developed a simple *flask* web application that has been extended to provide many different registration scenarios. This way, we are in control of both the crawler and the website it is attempting to crawl and can perform reproducible tests for most of the functionality.

The test cases contain login forms, registration forms, navigation, CAPTCHA solving, multi-stage forms, email verification, and other minor features. Altogether this covers 77% of all the code intending to cover at least 90% of it, leaving untested only the parts that strongly integrate with external services. In the continued development of the project, we suggest at least adding tests for all the newly introduced functionality.

Chapter 5

Deployment

For the final evaluation, we planned a repeat of the original experiment of crawling the top 1M websites by Tranco [14] to enable the comparison against the results of the crawler's past versions. The crawl time for a single site averages about 1.5 minutes so that the overall computational time can be measured in years, therefore requiring a high degree of parallelization to be feasible.

5.1 Distributed crawling

We have used an improved version of the distributed crawling architecture introduced by Kast [11]. The crawl was performed on the same hardware configuration that was used in March 2021, a single server with an 80 core Intel(R) Xeon(R) E7-8870 CPU and 512 GB of RAM. Instead of using the Proxmox hypervisor, we decided to use a bare-metal installation of Ubuntu Server 20.04. Through experimentation, we observed that the hardware was capable of running 50 Docker containers running the crawler in parallel.

5.2 Credentials

We used the same set of fictional credentials for all the websites in this crawl, except for the email address, which was generated for each website separately. The fictional person was chosen to reside in Germany on an existing street and city but a nonexistent street number. We did this to make our credentials look as realistic as possible without causing harm to any individual.

We used the API provided by Vonage [1] to acquire a phone number (currently Belgian due to legal restrictions in Germany). Currently, we have only done

[1] https://www.vonage.com/

a manual review of the received messages, but the API can in the future be used to implement automatic phone number validation.

5.3 Proxy vs. VPN

Since Switzerland is not a part of the European Union and therefore not bound by the GDPR, we required our traffic to originate from an IP inside the EU. Germany was chosen for this, the same as in the March crawl, as the legal research for which EnfBots are used focuses on German privacy regulations.

Usage of VPN is problematic since the IP address you receive from it is shared and well known, meaning many websites block it completely or just do not allow registrations from it (e.g., Linode). To avoid being blocked, we explored many different providers of rotating proxies, which could allow our 60 parallel crawlers to share a pool of IP addresses that were reserved just for our usage. Unfortunately, the market fragmentation meant the choice of a provider became complicated.

The requirements for our proxy server were as follows:

- support for geolocation in Germany
- support at least 16 concurrent requests
- $>$ 100 million requests per month (average of 130 per website)
- high bandwidth allowance (average usage 3MB per crawled domain)
- $>$ 10 concurrent IP addresses
- preferably residential IP addresses

The approximate cost of such a package ranged from 100€ per month for data center IPs to up to 6000€ per month for residential IPs due to high bandwidth requirements. In the end, we decided that for this crawl, due to the high price and additional configuration complexity, proxy services were not yet required. To further lower the percentage of blocked websites, either a suitable commercial proxy service provider needs to be employed, or the crawl could be performed with the aid of an EU research institution lending their IP address pool for the experiment. So we fell back to using a VPN, first attempting to use ProtonVPN[2], but ran into problems due to the VPN client's lack of support for headless machines. ExpressVPN[3] proved to be a solid alternative able to provide needed functionality, including geolocation in Germany.

[2] https://protonvpn.com/
[3] https://www.expressvpn.com/

Chapter 6

Results

The crawl took place between the 23rd of December 2021 and the 9th of January 2022. 611,351 randomly chosen pages from the list were processed during the interval. Comparatively, the crawl from March 2021 by Kast [11] covered the entire Tranco top 1M list. Note that despite a different generation time of the Tranco lists, the results should be comparable since it is the goal of Tranco to provide crawl lists for reproducible research. Since the crawler is still in active development, the primary motivation for the run was not to gather data for legal analysis but rather to test the performance of the crawler and collect the data required for future development. We can reconstruct large datasets for form classification, page classification, and submission validation from the obtained data. The results of the crawl also serve as a good indication of which modules of the crawler require the most attention.

6.1 Running time

The average running time per successfully loaded domain was 1.5 minutes, leading to a maximum of up to 50,000 domains crawled per day on our setup. This is comparable to the crawling speed reported by Kast, even though the updated version of the crawler can potentially visit more pages of a single website. In Fig. 6.1, you can observe the daily progression of the crawler. The observed slowdown between the 3rd and 6th of January was caused by increases in the email verification timeout and a reduced number of containers caused by a failed redeployment. The most representative data points are dated from the 7th and 8th of January, when the crawler processed 45,507 and 46,386 sites, respectively.

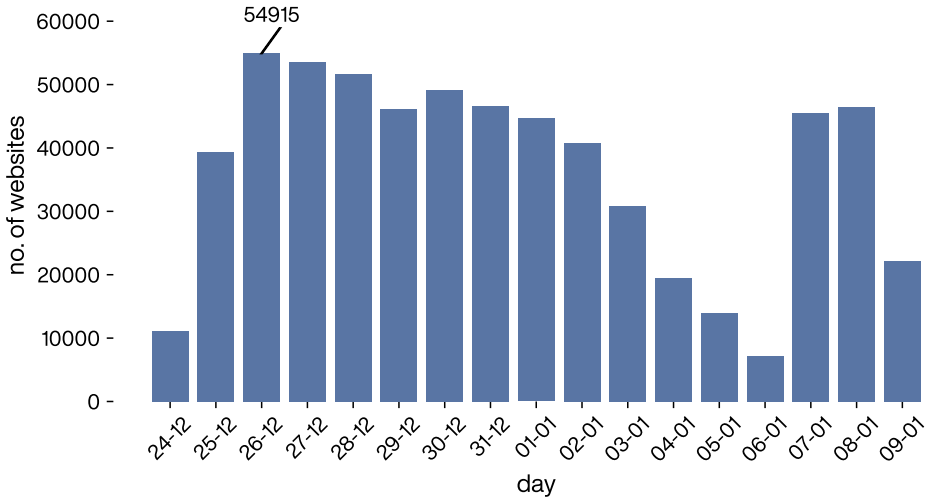


Figure 6.1: Crawl progress by day for its entire duration.

6.2 Load success rate

Loading the index page at the beginning of the crawl is not always successful. Fig. 6.2 presents the distribution of different scenarios that can occur. Out of the 611,351 attempted sites, the load of the index page was successful only on 415,066 (70,3%) domains. The most common problems were timeouts and TCP connection errors. We suspect two causes of timeouts: the website might be down at the moment (overburdened or broken), or we might be getting blocked by a bot detection system. On the other hand, we were able to determine that 87% of the TCP errors were caused by missing DNS records, while the rest were caused by bad server configuration. HTTP errors were less common; a manual analysis concluded that the biggest culprits were bad configuration and again various bot detection systems (e.g., Cloudflare).

Estimating how often we get blocked by anti-bot systems is hard because the results are often indistinguishable from a broken website configuration. We are not the direct target of these systems because we crawl each site slowly on a single thread and do not interact with it for more than a few minutes. Most protection systems are focused on preventing large-scale parallel crawls, spam, or DDoS attacks on a single page, which is far from our mode of operation, so we do not expect to be blocked based on our behavior.

In the study done by Le Pochat et al. [13] in April of 2019, they have observed that 85% of the websites could be successfully loaded. While the share of HTTP and TCP errors we encountered are within the margin of error (4% and 10%, respectively), we have detected a significant increase in the number of timeouts. It is possible that Le Pochat et al. used a longer timeout than

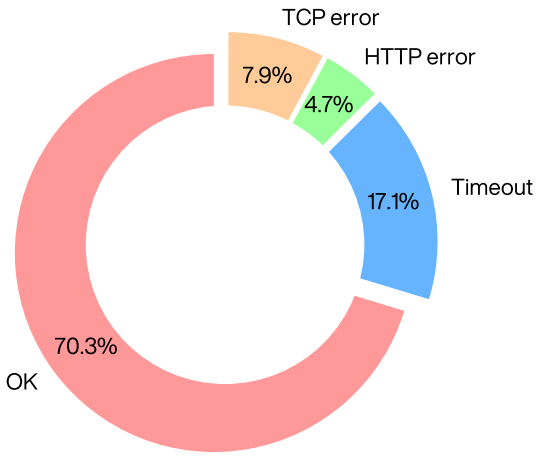


Figure 6.2: Observed index page load status.

40s, which we assumed to be enough for any working webpage to load. Another reason for the disparity can be our usage of a public VPN provider with known IP addresses, which many websites block by default. A recrawl of the timed-out domain from a private IP using cURL (no browser) has detected a timeout on only 11.5% of the pages that timed out in the main crawl of our study. This supports the hypothesis that the VPN is causing the timeouts since a non-IP-based bot detection system would block both the Selenium browser and a cURL without a spoofed User-Agent. We have already discussed solutions to this problem in Section 5.3.

In the following section, we will only be analyzing the websites where the index page has been successfully loaded, reducing the size of the dataset to 415,066 sites.

6.3 Registration status detection

The results of the crawls show that a registration form has been detected on 21.3% of all successfully loaded websites. To detect a registration form, the navigation module needs to find the subpage where the form exists and the classification module needs to classify it correctly. It is hard to estimate how many websites allow registration accurately. A manual analysis of 100 random English sites from the Tranco list has shown 28 sites contain any sort of registration form, of which 17 were also successfully detected by our crawler. On six sites, the detection failed due to the lack of support for forms in a popup dialog, three registration forms were missed due to incorrect language detection, and on the last three sites, the detection failed for more complex reasons. The sample size of this experiment is too small to draw any serious conclusions.

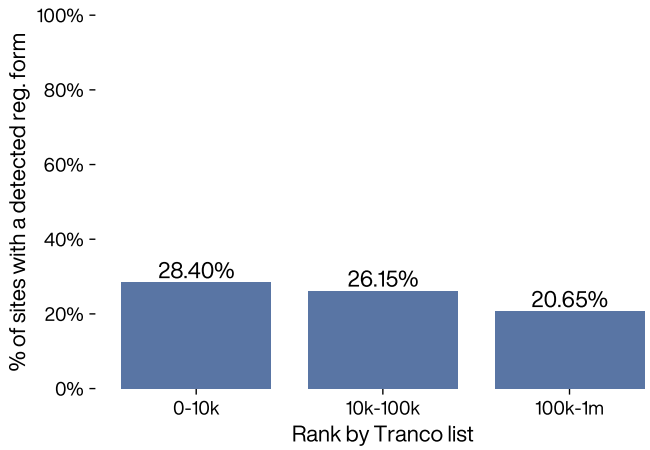


Figure 6.3: Form detection success in across popularity categories.

We can also compare the detection rate to the previous version of the crawler by Kast [11] which managed to detect a registration form on 11% of all websites. This is a big improvement, especially since we also evaluate the form more strictly and discard many forms which the previous version would try to register on.

The detection mechanism still yields some false positives. A manual analysis of 100 random, successfully detected forms has shown that 84 have contained a registration form, while the rest have been misclassified. Of those sixteen misclassified forms, eight were contact forms, three were newsletter forms, and five were login forms. This shows another limitation of the crawler. Even though some of the misclassified websites do contain a registration form that the crawler would prioritize over the selected one (mostly the case with login forms), the crawler currently stops at the first "registration" form detected, making misclassification a big problem. While an improved form classification algorithm could solve this problem completely, it might also be advantageous to register on more than the first detected form, at least in situations when the prediction certainty is low.

We have observed a falling trend in the number of detected forms as we go further down the Tranco list, confirming our assumption that less popular pages are also less likely to offer member registration as you can see in Fig. 6.3. Contrary to the observations by Drakonakis et al. [6], who found twice as many registration forms in the lower end of the popularity ranking as they did on the most popular ones, the falloff we detected was not as severe.

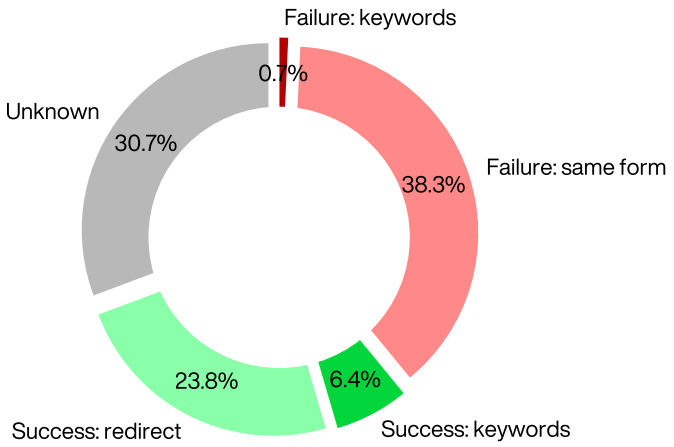


Figure 6.4: Results of the registration status detection pipeline.

Table 6.1: Comparison of the reliability of registration status indicators determined by manual review.

	True success	True failure	Misclassifed form
Success: keywords	28	0	2
Success: redirects	19	7	4
Failure: keywords	4	25	1
Failure: same form	2	25	3
Unknown	6	21	3

6.4 Registration status

The previous analysis leaves us with 88,464 pages on which a registration form was detected. Our submission success detection algorithm from Section 3.6 classified the attempts at registration on these pages into five categories, two representing success (determined by the keywords in the response message or by redirect), two representing failure (determined by the keywords in the response message or by detecting the same form), and the remaining unknown results.

We have detected a successful submission in 30.2% of forms, a failed one in 39%, while the remaining 30.7% submissions resulted in an unknown state. We have manually reviewed 30 submission attempts for each category. We present the findings of this analysis in Table 6.1. Note that for the misclassified forms, the detection mechanisms fail in undetermined ways so that the results might be biased due to misclassifications.

In 38% of cases, we are presented with the same form after submitting. There are multiple reasons this can happen.

- The form was not filled out correctly. This can be caused by a bad input format in some fields, invalid age selection, etc. It also happens for websites that require a credit card to register because we do not provide one at the moment. Note that not registering for paid services is the desired behavior. In the future, we can add a way to detect which field caused the error (it is usually indicated by the page) and resubmit the form each time we think we managed to fix an error.
- Solving the CAPTCHA failed.
- The form was misclassified. E.g., if we try to register on a login form, we will always be presented with the same form since our credentials do not match.
- Rarely, the same form is shown even though registration was successful.

According to manual review, the detection of the same form almost always means a failed submission, with just two out of 30 analyzed websites not removing the form from the site after a successful registration.

Failure detection through keyword analysis is the least likely scenario at just 0.7%, since an error is almost always accompanied by the form itself. Success keywords are detected more commonly, as was the case in 6.4% of all attempts. While this mechanism does not seem to produce false positives, we have noticed during the manual review that four pages displayed a success message, which remained undetected by the current implementation. This means that this part of the detection pipeline could definitely be improved.

Most of the successful registrations (23.8%) were actually detected by the simple redirect rule. The manual analysis does not completely support this indicator; almost a third of all redirects did not imply success. These sites mostly redirected us to external bot prevention services as CloudFlare[1], where the user has to solve a CAPTCHA in order to be allowed back to the website. The crawler does not currently support this flow. Adding the support would both improve the chances for successful registration and make redirect detection a more reliable way to indicate successful registration.

In 30.7% of the registration attempts, we have detected neither successful nor unsuccessful registration. We have already discussed different solutions to this problem in Section 3.6.5. Using the results of manual review, we can estimate that approximately 30% of all the undeterminate submissions were actually successful, which can also allow us to estimate the cumulative success rate of our crawler of just 32% of all detected forms.

Due to differences in the form detection, we cannot compare these results directly to the previous crawler version's results, but we can compare the

[1] https://www.cloudflare.com/

Table 6.2: The number of received emails compared to the detected status of registration
.

	Email received	No email
Success	9505 (36%)	17,188 (64%)
Failure	1676 (5%)	32,872 (95%)
Unknown	2910 (11%)	24,313 (89%)

overall success rate. In March 2021, the crawler registered successfully on 4% of all the crawled websites. Meanwhile, the current version was successful on an estimated 4.5% of all the pages (including the ones where the loading the index page failed). While this might not look like a significant improvement, we should note that the success detection techniques employed in Kast's work were even more unreliable, e.g., most of the submission attempts in the unknown category would actually be considered a success.

6.5 Received emails

Despite the fact that websites sometimes store your email address even if you do not actually submit the form [2], receiving an email to our generated email address is usually a good indicator of successful registration. We have received emails from 14,091 unique pages (3.4% of all successfully loaded and 2.3% of all attempted); of these, 8840 were recognized to contain an email verification link, which we used to confirm the newly created account. Another 3305 emails arrived after the email verification timeout had passed, so a verification attempt has not been performed. This indicates that the set timeout might be too low.

Not all websites send emails after a successful account creation, but this sets a stable lower bound on the success rate of our registration module. In comparison, the crawl by Kast [11] resulted in services sending an email in only 0.7% of all websites crawled. This result looks very promising.

In Table 6.2 we present a comparison of the number of received emails depending on the results of the submission validation oracle. It shows that the submission pipeline does a decent job of detecting failures because of the low amount of emails received in the category. According to Kubicek et al. [12] approximately 80% of the 701 web services involved in their studies sent an email. Considering this, we can assume that our success detection is too optimistic because we only received an email on 35% of websites we deemed successful. However, this can be partially limited by the short time of our study, as there are services that start sending, typically manually, emails only after a longer time.

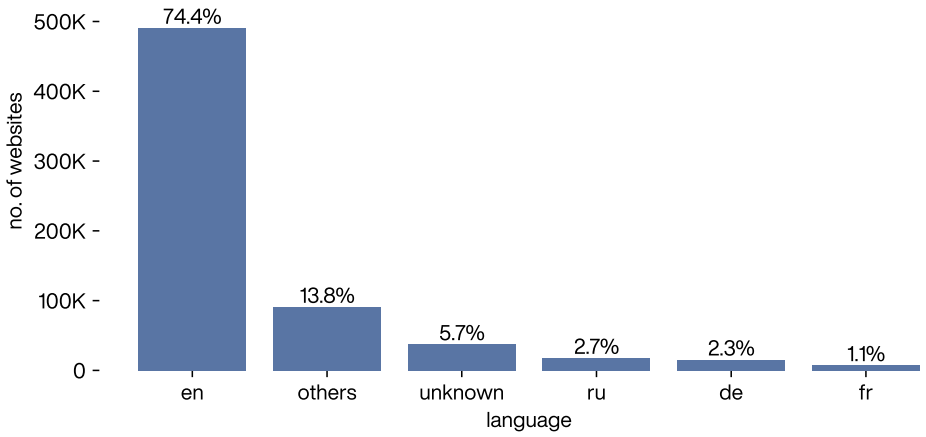


Figure 6.5: The proportion of different languages on the Tranco 1M list.

6.6 Received SMS messages

We also monitored the phone number we used in registration. Between the 23rd of December and the 9th of January, we have received 86 SMS messages. Because the crawler used the same phone number for all the websites, we cannot definitely know how many different websites these belong to (many messages are completely unidentifiable). We have successfully submitted our phone number to 31,797 websites, of which 6987 we deemed successful, which are both orders of magnitude higher. One reason for the disparity is the fact that many portals do not send any messages even though the registration requires a phone number. These services can keep phone numbers only as recovery/contact options. On the other hand, we have noticed that the phone number is often filled out wrong because of complications with number formatting and country codes, e.g., a phone number sometimes spans over multiple input fields, which we currently do not support.

6.7 Per language results

We observed over 74% of the most popular pages primarily in English, and another 3% allow switching to English. In the Fig. 6.5, you can observe the distribution of the languages after the crawler tried selecting a supported language. Because of the heavy prevalence of English, we focused most of our development on that language. The other languages are technically supported yet not optimized.

In this section, we will analyze how our detection and registration success rate depends on the detected language to evaluate the performance of less

Figure 6.6: Comparison of registration outcomes over different languages.

common languages. We will group all the languages below 1% popularity into a single group for conciseness. In Fig. 6.6, we can see that our intuition that English would perform best was not necessarily correct. For example, more forms get detected on Russian websites, with significantly more errors and also more successes as a consequence. A similar pattern can be noticed to a lesser extend for other languages too. This could be partially due to the underlying complexity distribution over different languages. The most complex websites almost always support English, while websites in other languages are often more local and simpler, matching more common registration form patterns. Additionally, this could also be caused by a more unreliable detection of failed submissions, which is also in part language dependent.

6.8 CAPTCHA systems

ReCAPTCHA is by far the most popular defense websites use to protect form submissions from bots. In Table 6.3, we compare the market share estimation of supported CAPTCHA providers between our results (*Enfbots 2*), the results of the previous crawl (*Enfbot 1*), and the estimation by BuiltWith[2]. Note that neither of these perfectly represents the true status. Our crawler severely underestimates the presence of CAPTCHA systems because it only looks for them on webpages with a registration form. The BuiltWith estimations are also inconsistent. E.g., it reports the market share of reCAPTCHA in general at 22%, while the individual shares of its different versions only add up to only 12%. In the table we present the individual shares.

[2] https://trends.builtwith.com/widgets/captcha

Table 6.3: The popularity of supported CAPTCHA mechanism as reported by different sources.

	BuiltWith	Enfbot 1	Enfbot 2	Success rate
reCAPTCHAv2	8.86%	1.16%	4.01%	33.10%
reCAPTCHAv3	2.69%	0.48%	0.96%	25.60%
hCAPTCHA	0.24%	0.01%	0.01%	39.02%
Image CAPTCHA	-	-	0.23%	23.50%

Table 6.3 shows that submitting the forms with hCAPTCHA has the same success rate of about 40% as forms without bot mitigation mechanisms. The most popular reCAPTCHAv2 reduces the success rate to 33%. Meanwhile, reCAPTCHAv3 seems to drop the success rate to only about 25%. This version assigns each user a score that corresponds to the probability of being a human. It is up to the developer's discretion to decide which threshold to use. The external solving service we use is limited in this aspect, as it can reliably produce responses with a score of only 0.3. Anything more that is a fortunate coincidence, so websites that demand higher scores result in bot detection.

The image CAPTCHA drops the success rate even further for two reasons. First, our detection of the CAPTCHA image and the corresponding input field is unreliable since there is no standardization involved here. Each website that uses a custom image CAPTCHA also has a custom layout that is hard to detect. Second, we noticed the wrong solution from the image CAPTCHA solving service, where completely valid images get incorrectly recognized.

Chapter 7

Related work

7.1 Automating registration or login

Large-scale analysis of security and privacy of user data in authenticated sections requires automating either the registration or login procedure. Drakonis et al. [6] have independently developed a registration crawler based on Selenium as a part of their investigation into cookie hijacking vulnerabilities. The core principles used are comparable to our crawler; namely, they also use keyword-based navigation, form classification, and input classification. Their crawler interacted only with registration forms and login forms, while ours is also able to detect newsletter and contact forms, which makes our implementation easier to generalize for other tasks. For comparison, they detected a signup form on 168,594 out of 1,585,964 (10.6%) pages from Alexa rankings. Of these, they reported a 13.7% success rate of automated registration. This brings their success rate to 1.38%. The crawlers comparison depends on the requirements for claiming a successful registration. Unlike Drakonis et al., we do not require a successful login using the registered credentials to confirm successful registration. Hence the reported success rate of 4.5% in Section 6.4 would be lower with the requirements from Drakonis et al. However, considering only the proportion of the website that sent us an email, we can estimate that our tool performs better at 2.3%, which would be even lower in Drakonis et al. This can be partially accredited to our support for CAPTCHA solving, multi-stage registration forms, and more advanced input classification methods.

Similarly, Chatzimpyrros et al. [2] developed a registration automation crawler designed to study the leaks of personally identifiable information in registration forms. The nature of their research allowed them to assume that any form containing two input fields and a submit button is a registration form. They were not concerned with a successful registration since they merely wanted to submit as much personal data as possible and detect any

potential leaks. This makes the comparison of their reported success rate of 26% with our 4.5% irrelevant, as they assumed all login and contact forms to be registration forms, and they never checked whether the submission was successful.

7.2 Other generic crawling frameworks

There already exist generic policy and privacy research tools, namely *OpenWPM* by Englehardt and Narayanan [8] and *webXray* by Libert [15]. They both facilitate the study of 3rd-party tracking and fingerprinting by allowing researchers to capture the user data the website submits to the server. While the goal of those projects is different from ours, the solutions they have devised can be adapted and used to extend the functionality of our crawler. We have already integrated some techniques of bot detection evasion used in OpenWPM (e.g., using Xvfb instead of the headless browser). WebXray has introduced a corporate ownership tracking mechanism that can be used to determine the legal entity representing both the website itself and any 3rd-party tracking tools used on the page.

Chapter 8

Discussion

8.1 Future considerations

We have presented many limitations with accompanying ideas on how to alleviate them in past chapters. Here we shall only summarize the most important high-level ideas.

We conclude from the results that the most significant improvement can be made in the module responsible for filling and submitting the form. More than half of all sign-up attempts fail in that step. While some failures are unavoidable (e.g., credit card requirements), many are technically feasible. Improving the error detection would help increase the success rate and make the results more sound since we would know which fields break the submission most often.

Once the datasets we have collected in this work are labeled, we can develop classification models for forms, page content, input fields, and submission status detection. Misclassification of those parts of a website usually completely stunts the sign-up attempt, so any improvement here can improve overall performance substantially.

In the rest of this section, we propose alternative and independent crawling goals that can be implemented easily because of the modular codebase.

8.1.1 Login

The ability to login into a portal could be added to the crawler similarly to what has been done by Drakonis et al. [6]). This would not require significant changes, as the navigation and form classification modules are already able to detect login forms. The feature could not only be used to verify registration, but it could additionally allow for mass-scale analysis of security and privacy of login mechanisms deployed on the internet.

8.1.2 SSO sign up

Zhou and Evans [21] have developed an automatic Single Sign On (*SSO*) registration tool SSOScan. An implementation of a similar module would allow our crawler to sign up using the most common SSO identity providers (e.g., Google and Facebook). This could be either the primary sign-up mechanism or just a fallback when email registration fails, depending on research needs. This feature would enable us to study how the choice of the mechanism affects email marketing and private data leakage on websites that support both SSO and email-based sign-up.

8.1.3 Modal registration dialogs

Many websites have their registration form hidden in a modal dialog that can be accessed by clicking a button in a navigation bar. The crawler currently only navigates using hyperlinks and only detects visible forms, therefore completely ignoring modal forms. In the manual analysis of 100 websites, we detected this scenario on six pages, indicating that it is common. Supporting this feature could, therefore, significantly improve the overall form detection reliability.

8.1.4 Automatic translation

Despite what the results from Section 6.7 show, we believe that there is a considerate disparity between the performance of the crawler on English sites and those in less common languages. The keyword-matching methods are very effective in the English language due to the lack of noun conjugation, but they often fail in other more complex languages. Using a translation service like Google Translate, we could first translate the string we are classifying into English and then use the English keywords to determine the correct category. This approach has already been proven by Drakonis et al. [6]. There are other benefits of machine translation.

- It requires training just a single ML model for any of the classification tasks, removing the need to develop NLP models in language we do not speak.
- It solves a commonly observed problem where a website is only partially localized in English. Forms and privacy policies often remain only in the native language. Using the translation module, we could translate only the missing parts and still rely on English keyword matching.

8.2 Conclusion

Automated interaction with authentication forms is a challenging problem due to the sheer variety of website implementations. There are no strict rules

for web development, just the best practices. The crawler we developed as the central part of this thesis can already distinguish many of these scenarios and react accordingly.

The modular approach we devised allows the crawler to be highly flexible in terms of its goals. By applying minor modifications, it can be used to create new accounts, sign up to newsletters, submit contact forms, or log into a website without knowing its structure beforehand. With the use of the crawler, many tedious manual studies of internet privacy and security could be automated and therefore conducted on a much larger scale than was possible previously.

Priority queue-based navigation proved to be an efficient method of quickly identifying the registration pages without resorting to BFS scan while remaining adaptable on sites where keyword-matching is ineffective. The traditional form classification algorithm we refined has a satisfactory accuracy but can also be later used as a baseline for machine learning models. The novel approach to reCAPTCHA integration also proved to be highly reliable, having encountered no pages where the detection itself is the problem. While the accuracy of the form autofill module and submission detection oracle still leaves much room for improvement, their current versions already perform well on simpler forms.

By crawling more than half of the Tranco list of 1M most popular websites, we achieved a significant improvement upon the previous version with a 20% detection and an estimated 6.4% success rate when creating new accounts. By analyzing the results, we have accurately determined the weak points of the current implementation and identified many new potential avenues for future development. The dataset collected can be used to develop machine learning models to provide a more accurate classification of forms, input fields, and pages themselves, which will, in turn, improve the overall reliability of the entire framework.

Bibliography

[1] Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, and Michele Bugliesi. Testing for integrity flaws in web sessions. In *European Symposium on Research in Computer Security*, pages 606–624. Springer, 2019.

[2] Manolis Chatzimpyrros, Konstantinos Solomos, and Sotiris Ioannidis. You shall not register! Detecting privacy leaks across registration forms. In *Computer Security: ESORICS 2019 International Workshops, IOSec, MSTEC, and FINSEC, Luxembourg City, Luxembourg, September 26–27, 2019, Revised Selected Papers*, page 91–104, Berlin, Heidelberg, 2019. Springer-Verlag. doi:10.1007/978-3-030-42051-2_7.

[3] Pupeteer contributors. Pupeteer, headless Chrome Node.js API. URL: https://pptr.dev/.

[4] Selenium contributors. Selenium, web browser automation. URL: https://www.selenium.dev/.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding, 2019. arXiv:1810.04805.

[6] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1953–1970, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417869.

[7] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. I never signed up for this! Privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies*, 2018, 01 2018. doi:10.1515/popets-2018-0006.

[8] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM CCS 2016*, 2016.

[9] 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. URL: http://data.europa.eu/eli/dir/2002/58/oj.

[10] 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, 2016. URL: http://data.europa.eu/eli/reg/2016/679/2016-05-04.

[11] Patrice Michael Kast. Enforcement bots: Nothing can block us! Automating website registration for GDPR compliance analysis. Bachelor's thesis, ETH Zurich, 2021.

[12] Karel Kubicek, Jakob Merane, Carlos Cotrini, Alexander Stremitzer, Stefan Bechtold, and David Basin. Checking websites' GDPR consent compliance for marketing emails. *Proceedings on Privacy Enhancing Technologies*, 2022.

[13] Victor Le Pochat, Tom Van Goethem, and Wouter Joosen. Evaluating the long-term effects of parameters on the characteristics of the Tranco top sites ranking. In *Proceedings of the 12th USENIX Conference on Cyber Security Experimentation and Test*, CSET'19, page 10, USA, 2019. USENIX Association.

[14] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019. URL: http://dx.doi.org/10.14722/ndss.2019.23386, doi:10.14722/ndss.2019.23386.

[15] Timothy Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *CoRR*, abs/1511.00619, 2015. URL: http://arxiv.org/abs/1511.00619, arXiv:1511.00619.

[16] Dheeraj Mekala and Jingbo Shang. Contextualized weak supervision for text classification. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 323–333, Online, July 2020.

Association for Computational Linguistics. URL: https://www.aclweb.org/anthology/2020.acl-main.30.

[17] Panagiotis Papadopoulos, Nicolas Kourtellis, Pablo Rodriguez Rodriguez, and Nikolaos Laoutaris. If you are not paying for it, you are the product. *Proceedings of the 2017 Internet Measurement Conference*, Nov 2017. URL: http://dx.doi.org/10.1145/3131365.3131397, doi:10.1145/3131365.3131397.

[18] Suphannee Sivakorn, Jason Polakis, and Angelos D Keromytis. I'm not a human: Breaking the Google reCAPTCHA. *Black Hat*, pages 1–12, 2016.

[19] Oleksii Starov, Phillipa Gill, and Nick Nikiforakis. Are you sure you want to contact us? Quantifying the leakage of PII via website contact forms. *Proceedings on Privacy Enhancing Technologies*, 2016(1):20–33, 2015. URL: https://doi.org/10.1515/popets-2015-0028, doi:doi:10.1515/popets-2015-0028.

[20] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: Using hard AI problems for security. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 294–311, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[21] Yuchen Zhou and David Evans. SSOScan: Automated testing of web applications for single sign-on vulnerabilities. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 495–510, USA, 2014. USENIX Association.

Appendix A

Appendix

A.1 Form annotation interface

The most time-consuming part of any dataset preparation is the manual labeling process. In our case, that was required to train the form classification models, so we developed a simple graphical interface to facilitate the process. As displayed in Fig. A.1, it presents the user with a screenshot of the form in question, a screenshot of the entire website, and the HTML code of the form itself to give the annotator the best chance of correctly determining the type of form. Because some websites contain multiple forms close by, the form we are annotating is always surrounded by a red box. The exact URL where we collected the form is also displayed and clickable to allow the user to visit the page if some context is missing (e.g., sometimes the screenshots are misaligned).

The user then has to select one of the 6 form categories and any additional tags that fit the form (e.g., multi-stage, modal). Any form can also be skipped if it cannot be classified (e.g., completely hidden forms) or if it is malformed in any way (e.g., form in a form). All the interaction can also be done using keyboard bindings to speed up the process significantly.

A.2 reCAPTCHA solver

The central part of our reCAPTCHA detection and the solving process is our implementation of the reCAPTCHA Javascript API. We replace the real version when a website attempts to load it.

To understand how it works, we must first know how reCAPTCHA can be integrated into a website.

First, you need to register with the reCAPTCHA service and receive a site key that will be used to add the reCAPTCHA to your website and a secret

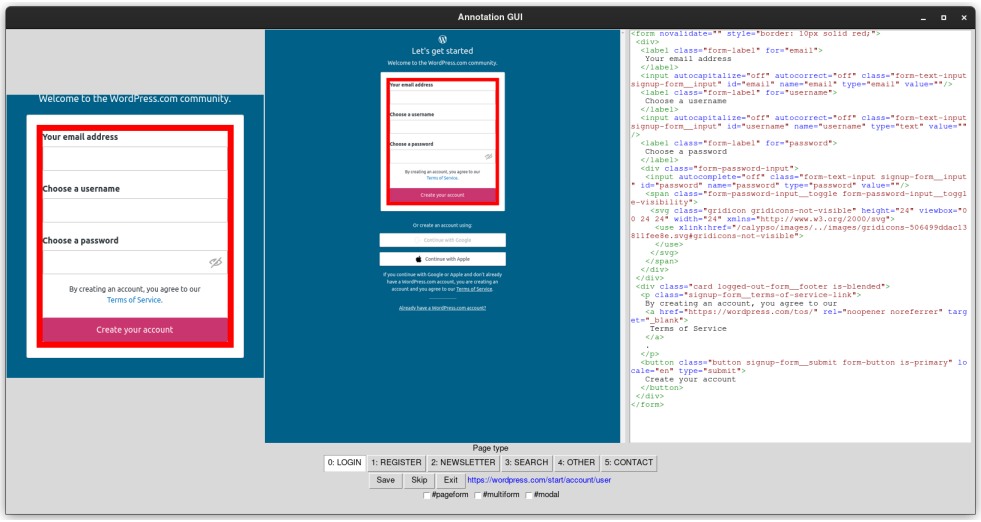


Figure A.1: Screenshot of the graphical interface for form annotation.

key that can be used to verify the reCAPTCHA response.

Next, you import the reCAPTCHA JavaScript on the website containing the form you would like to protect directly from Google's CDN. An important detail here is that you cannot host the reCAPTCHA implementation yourself because it contains constantly changing URLs that are used to interact with the backend service.

When the script is set up, the easiest way to include a reCAPTCHA field into your form is to add a container `div` inside the form with the class "g-recaptcha" and a `data-sitekey` attribute containing the site key. When the page and the reCAPTCHA JavaScript are loaded, a reCAPTCHA checkbox will be rendered inside an `iframe` inside the `div`. A hidden `textarea` with the name "g-recaptcha-response" is added to your form. We show an example of this integration in Listing 1.

When the CAPTCHA is solved, the "g-recaptcha-response" `textarea` will be filled with a unique solution token that can be verified on your backend server using the secret key you received in the first step.

The widget can also be rendered programmatically using the global `grecaptcha` object provided by the reCAPTCHA library. On page load, you call the `grecaptcha.render` method with all the parameters and a container where the widget should be rendered. This will create the same "g-recaptcha-response" `textarea` and the `iframe`. You can also retrieve the solution token by calling the `grecaptcha.getResponse` after the CAPTCHA has been solved if the form submission is made using AJAX.

As can be seen from the way reCAPTCHA is integrated, the only thing we

```
<form action="?" method="POST">
 <div class="g-recaptcha" data-sitekey="your_site_key"></div>
 <input type="submit" value="Submit">
</form>

<form action="?" method="POST">
 <div class="g-recaptcha" data-sitekey="your_site_key">
  <iframe title="reCAPTCHA"
   src="https://www.google.com/recaptcha/api2/anchor?b={nonce}">
  </iframe>
  <textarea id="g-recaptcha-response" name="g-recaptcha-response">
  </textarea>
 </div>
 <input type="submit" value="Submit">
</form>
```

Listing 1: simplified reCAPTCHA widget integration

need for our submission to be successful is a valid response token.

This will be obtained from our external CAPTCHA solving service; we only need to provide them with the parameters used to render the CAPTCHA, the User-Agent we are using, and the domain this CAPTCHA is present on.

The backend server can also verify that the response token was issued for the same IP used to submit the form. In our case, these differ (CAPTCHA is solved externally and used by us). We could alleviate this problem by providing the solving service with the same proxy server that we are using to get a token that matches our IP. The EnfBot crawler supports this functionality, but since the final crawl was using a VPN instead of a proxy service, it has not been used in it. Luckily, most websites do not actually check if the token was issued for the same IP that performed the submission, so this is not a severe problem.

A.2.1 reCAPTCHA stub

The only part keeping us from solving this type of CAPTCHAs is the collection of initialization parameters. If the website is using the simple reCAPTCHA class name-based initialization, this is easy; we can scan the DOM for the existence of an element with a "g-recaptcha" class name and extract the parameters from it. But if the reCAPTCHA widget has been rendered by interacting with the JavaScript API, this becomes more problematic.

Initially, we processed all the JavaScript files loaded on the page using regex, but this proved unreliable. The regex scanned for all instances of "sitekey" assignment, but as you can see in Appendix A.2.1 this can quickly fail.

```
// works:

grecaptcha.render('html_element', {
  'sitekey' : 'your_site_key'
});

// Does not work, the site key is
// extracted as string "mykey" instead of its value

mykey = "yoursitekey"
grecaptcha.render('html_element', {
  'sitekey' : mykey
});
```

Listing 2: Example for site key detection

Instead, we implemented a stub for the entire reCAPTCHA library that captures any request made to the known URLs which contain the reCAPTCHA scripts and substitutes the responses with its own implementation of the reCAPTCHA API. Our stub provides the webpages the same "grecaptcha" global object as the authentic version, but instead of rendering the CAPTCHA widget, it just stores the configuration parameters that our crawler can later access. The stub also creates the hidden "g-recaptcha-response" textarea and provides the `getResponse` method so the webpage we are crawling cannot notice the difference between them.

Now we are ready to construct the entire workflow of the CAPTCHA solving modules:

1. stub is loaded
2. stub scans for the "g-recaptcha" configuration elements and stores the parameters
3. potentially the website calls the `grecaptcha.render` function on the stub who stores the call parameters
4. we render the output `textarea` on the page
5. the crawler gets the parameters from the stub and submits them to the solving service to receive the response
6. the crawler gives the response back to the stub, which fills the textarea and potentially answers a `getResponse` call

ETH

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):

DESIGNING A GENERIC WEB FORMS CRAWLER TO ENABLE LEGAL COMPLIANCE ANALYSIS OF AUTHENTICATION SECTIONS

Authored by (in block letters):

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

Name(s):	First name(s):
LODRANT	LUKA

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	Signature(s)
ZÜRICH, 17.1.2022	

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