Analysis of the Information Security Job Landscape in Austria

Markus Reiter

University of Innsbruck, Austria

Abstract. In this paper we present a survey conducted by scraping and analysing various job search websites in order to get an overview of the current situation and trends regarding the information security job landscape in Austria.

Keywords: Information Security · Scraping · Job Advertisements

1 Introduction

Information security management is more and more becoming an essential area of expertise for large and small companies alike. In order to get a better understanding of what the information security job landscape looks like in Austria, we conducted a survey of three job search websites popular in Austria: *Monster* [1], *Indeed* [2] and *StepStone* [3]. Our survey is comprised of three main steps: Firstly, we needed to scrape data each of the three job search websites. As will be explained in more detail in Section 2, we evaluated various scraping tools before we advanced to the actual implementation of the scraping process which will be explained in Section 3. The second part of our survey was then to use natural language processing in order to generate a more or less structured version of the information contained in the mostly unstructured text of the job advertisements. Lastly, we analysed and visualised the output data from the natural language processing step in order to fulfil our objective of forming a good overall perspective of the cybersecurity job landscape in Austria. In Section 4 we will present the findings we made during this process.

2 Evaluation of Scraping Tools

The first step towards being able to analyse job advertisements on the internet is being able to properly retrieve them. For this reason, we evaluated various online scraping services.

The first contender we found was *OctoParse* [4], which provides a desktop application for scraping. Its free tier allows scraping unlimited pages on unlimited computers, with two concurrent scraping jobs. It allows configuring 10 different crawler configurations. There was, however, one major drawback to using *OctoParse*: Its slogan claims "Easy Web Scraping for Anyone" – "Anyone" in this context meaning anyone with a Windows computer, since the desktop application is only supported on Windows. Linux and macOS users would have to run a virtual machine in order to use it, which is definitely not a viable solution for scraping a few websites. So we began looking for alternatives.

One alternative we found was *ScrapingBot* [5]. Unlike *OctoParse*, which provides a desktop application, *ScrapingBot* does not provide an application at all. It is meant to be used similarly to using a proxy server. You can make an HTTP request with the same URL you normally would, and it responds with the contents, which includes any rendered JavaScript. Its free plan includes 100 API calls per month, therefore this also was no viable solution either, since the API calls would probably have been used up before the scraper was even working properly.

The next service we found was *ScrapeStorm* [6], which is very similar to *OctoParse* in terms of what it provides. It offers a desktop application which

works on Linux, macOS as well as Windows. In that regard, it was already looking like a better solution *OctoParse*. The free plan allows 10 different crawler configurations and 1 concurrent local run. It also allows 100 exported rows per day, either into a file or directly into a database. All in all, *ScrapeStorm* seemed like a good fit.

After choosing *ScrapeStorm*, we started by setting up the crawlers. The websites we wanted to scrape are `monster.at` [1], `at.indeed.com` [2] and `stepstone.at` [3].

For each website, we would have to implement two steps. First, we need to scrape the list of search results, which provides us with a list of links to the detail pages for the job advertisements. The second step is then to scrape the detail page itself.

As a first step, we tried scraping a detail page for a single job advertisement on `monster.at`. Fortunately, this immediately highlighted a major problem at the beginning of the scraping process: A job detail page on `monster.at` always includes the actual content of the advertisement inside of a nested `iframe` element. Using *ScrapeStorm*, we were able to select that `iframe` element, but we failed to extract any data or text from it. The only thing we got was an empty string. This meant that we had to abandon the idea of using *ScrapeStorm* for our purposes.

With this setback, we now had to find another way to scrape this data. After taking a second look at the *ScrapingBot* homepage, we saw that it is using headless *Chrome* for its back-end. We decided to venture further into this direction and found *Selenium* [7], which can be used to programmatically use a web browser.

Selenium supports all of the major web browsers, the only thing you need is the so called "webdriver" for the corresponding browser. Additionally, *Selenium* can be used with a wide array of programming languages. For our purposes we chose to go with the Ruby version of *Selenium*. Specifically, we chose to use *Rake* [8], which is similar to *Make*, since it allows us to use the full functionality of *Ruby* [9] and specify tasks for each website. These scraping tasks can then be run with a simple `rake` command.

Given the setback we encountered on `monster.at`, the first thing we tried with *Selenium* was to implement a simple task to extract the contents of the nested `iframe` element in their detail page layout. The first thing we needed to do was to fetch a job page. Next, we needed to locate the nested `iframe` element; luckily this element is marked by a unique `id` field called `JobPreviewSandbox`. Once we had saved the nested `iframe` into a variable, we could use the `switch_to` function provided by *Selenium* to change to it from the main frame.

```
iframe = find_element(id: 'JobPreviewSandbox')
switch_to.frame(iframe)
```

Seeing this proof-of-concept task working, we finally settled on using *Selenium* and *Rake* together with the *Firefox* web browser.

3 Implementation

In this section we first present the implementation of the scraping step for each of our chosen websites. After that we will go into more detail on how we further processed the scraped data and will also mention some specific things to look out for when extracting information from that data. Finally we will also highlight the tools we used to analyse and visualise the collected data.

3.1 Scraping

As already briefly mentioned in Section 2, our goal was to implement scraping for *Monster* (`monster.at`), *Indeed* (`at.indeed.com`) and *StepStone* (`stepstone.at`). In this section we will go into more detail about the exact steps which were necessary to reliably scrape each respective website.

Monster The first thing we needed to determine for *Monster* [1] was how to perform searches. For this, we simply made a search request manually in a web browser. With the search string "information security" immediately noticed that it was escaped as `information-security` rather than the standard way to escape HTML, i.e. `information%20security`. We needed to consider this when building the query URL.

Furthermore, we needed to assess how search results are paginated. *Monster* does not use a classic pagination method using query string parameters, but rather uses a single "infinite" list which loads new search results automatically when you scroll to the bottom, or alternatively, click the "load more jobs" button.

With *Selenium*, it was quite easy to extract all URLs leading to detail pages from the result list. Loading more than the first result page was, however, quite convoluted. Simply scrolling down was not a good option, since then there would be no good way of knowing when to stop scrolling. Conveniently, *Selenium* provides function to scroll to a given element, so we simply tried selecting the "load more jobs" button and scrolling to it. This is where we hit another problem. We could scroll the the button, but we were not able to click on it due to a pop-up blocking exactly the lower part of the page where the button is located, which meant we first had to dismiss this pop-up, which further meant introducing another potential point of failure, but for the moment, this solution was working.

In essence, the workflow for parsing URLs was as follows:

1. Perform search request.

2. Locate pop-up and dismiss it.
3. Locate "load more jobs" button.
4. Scroll to "load more jobs" button if it exists, otherwise go to step 6.
5. Click "load more jobs" button.
6. Select all result URLs and collect them into an array.

This workflow seemed to work as expected, however, we later encountered spurious error between step 4 and 5. As mentioned previously, new search results are automatically loaded when you scroll to the bottom of the page. Obviously, scrolling to the "load more jobs" button was triggering that functionality and render the reference to the button invalid, making step 5 fail. While searching for a solution, we came upon *Selenium's* `execute_script` function, which can be used to directly execute *JavaScript* code in the current browser window. This function enabled us to directly click the "load more jobs" button without scrolling to it, which also made it possible to eliminate the step for locating and dismissing the pop-up.

The final workflow now looked like this:

1. Perform search request.
2. Locate "load more jobs" button.
3. Click the "load more jobs" button if it exists, otherwise go to 4.
4. Select all result URLs and collect them into an array.

For the second part, the detail page of the job advertisement, we could now loop through all URLs and extract information from each one. On *Monster*, each detail page contains the job title and a subtitle. The subtitle usually contains the location of the open position. Additionally, there is a sidebar which contains further metadata like the contract type, category, career level and publication date. The main content of the job advertisement is embedded into a nested `iframe` element. As mentioned in Section 2, using *Selenium*, we could easily extract the content of this `iframe` element using the `switch_to` function. All other data we could extract equally easily by selecting the corresponding elements using CSS selectors.

Indeed For *Indeed* [2], the initial step to implement proper scraping was to implement fetching search results. Luckily, unlike on *Monster*, the search string could be escaped as `information%20security` using standardised HTML escape codes.

Pagination on *Indeed* is using a query parameter called `start` which indicates the offset from the first result. Additionally, the pagination buttons are located at the bottom of the results page. With the "next page" button acting as the stop indicator we can trivially implement the gathering of all detail page URLs as described below:

1. Initialise empty array.
2. Perform search request with offset `0`.
3. Select all result URLs and add them to the array.
4. If "next page" button exists, repeat step 2 with offset + `10`.

Similarly to *Monster*, the detail pages on *Indeed* contain the title of the job advertisement as well as a metadata section. Care has to be taken when extracting data from this metadata section, since not every detail page contains every entry. Most pages seem to contain the location, but only some contain the contract type of the open position. For this reason, we simply treat all of these as optional and ignore any `NoSuchElementError` when searching for the respective element. In contrast to *Monster*, the main content of the advertisement is contained as plain text, so no special treatment is necessary in this case.

StepStone Implementing the gathering of search results on *StepStone* [3] was largely similar to the workflow used for *Indeed*. The search string can be escaped using the standardised HTML escape codes, furthermore there are some query parameters we make use of: The `fu` parameter can be used to set the job category. We used `1000000` for this parameter to specify the "IT & Communications" category. For faster scraping, we also set the `li` parameter to `100` in order to display 100 results per page rather than the default 25. Finally, we use the `of` parameter to specify the result offset which is then used in the same way as for `Indeed`. *StepStone* displays the pagination controls at the top of the results page. These include the current as well as the last page number. Using this information we can stop the loop once we reach the last page.

A detail page on *StepStone* contains the job title at the top, followed by a metadata section. The metadata contains the location, type of contract and type of employment (i.e. full-time or part-time). The main content of the job advertisement, similarly to *Indeed*, is a section containing plain text. This makes it trivial to extract all data.

Storage of Scraped Data Now that we had successfully built the foundation for properly scraping all websites, the last step was to store the scraped data in a format which can be used for further processing. Using a full-fledged database solution, e.g. MySQL or Postgres, was out of the question from the start. Setting up a dedicated database server for what is – relatively speaking – a very small amount of data would add too much maintenance overhead. Another aspect to consider is the data layout. In database terms, we need to store only a single table of data, which was another reason why we decided against using a database for storing the collected data altogether. This included embedded databases like SQLite which consist of a single file and don't need a dedicated server and

client. Ultimately we decided on storing each scraping result in a separate *JSON* file. *JSON* is heavily used for APIs on the web which means that virtually every programming language supports deserialising and serialising data in *JSON* format. To avoid duplicates, we chose to save the files as `<website>-<id>.json` where `<website>` is the name of the website, i.e. `monster`, `indeed` or `stepstone`. For the `id` we calculate the SHA2 checksum of the URL.

For example, the scraped data from the URL `https://at.indeed.com/rc/clk?jk=db6f5459c93e648c&fccid=1272531993b16790&vjs=3` would be saved as `indeed-43da336235334055a25f32a55e7cd343128905d782cd749349659ca3d3d38174.json`.

Removal of Duplicates After a couple of scraping runs, we noticed that there were duplicate results. This happened for all three websites. These duplicates were either completely equal or had only minor differences in their HTML source code. Nevertheless, we needed to introduce a post-processing step after the scraping is finished. We implemented a new *Rake* task called `deduplicate` which reads all results and sorts them by date. We then convert the results' title and HTML body content to plain text and group the results according to that. Finally, we can find all duplicates by selecting all results with more than one entry in the corresponding group and remove all but the latest one.

3.2 Natural Language Processing

Once the scraping was done, we could focus on extracting information from the scraped data. For the purpose of data extraction, we chose Python. Like Ruby, Python is an interpreted and dynamically typed language, which makes it a good candidate for natural language processing since we can iterate over code changes very quickly. We chose Python over Ruby due to the vast amount of libraries and frameworks available dedicated to natural language processing, first and foremost the Natural Language Toolkit (NLTK), arguably one of the most widely used platforms for natural language processing. It contains a great variety of tools including classification, tokenisation, stemming, tagging, parsing and semantic reasoning. [10]

Data Cleaning The first step when performing any sort of natural language processing should be data cleaning. Specifically, this means removing irrelevant data and bringing the data into a format that can be processed more easily than the raw source. In our case the most important part that needed cleaning was the body text of each result. Each result contains the raw HTML source code of the body text, so the first step was to strip the text from any contained HTML tags. We used the aptly named `html2text` Python package for this purpose. After further inspection of the results, we noticed that some contained formatting,

links and images in Markdown format. We quickly realised that `html2text` is in fact an HTML to Markdown converter instead of a HTML to plain text converter as we previously deemed from the name. Luckily, `html2text` provides `ignore_emphasis`, `ignore_links` and `ignore_images` options which can be used to remove the Markdown output for these elements. Once we had the plain text version of the body content, we could subsequently use the `word_tokenise` function from the NLTK to split the text into tokens to make further processing easier.

Extracting the Salary The first data point we decided to extract was the salary for the advertised job. Even with only this single data point, there is a lot of variation in how the salary is represented.

43,595 Euros	EUR 2.600,00	65,000 €
30k Euros	70.000 €	€60,000
3,100.00 EUR	€ 4.209,-	€85k
EUR 2.800,-	€ 3.933,0	€3.007,20
EUR 50k	€ 55.000,–	80.000 Euro
EUR 5700	€ 65.000,—	60000 €

Fig. 1. An incomplete list (extracted from actual data) of number formats used to represent salary.

As seen in Figure 1, the diversity of number formats is quite extensive. To explain the steps we took to further clean the input data, we will define the `clean_text` step as cleaning the complete text before it is tokenised and the `clean_word` step as cleaning a single word after tokenisation.

The first step we took was to add another sub-step to our existing `clean_text` step in which we used the regular expression `/(EUR|€)/` and replace it with `' \1 '`, i.e. we add an extra space around `EUR` and `€` to cover the case where there is not already a space between the currency and the amount. We don't care about introducing extra spacing when there already is a space since the tokenisation will remove it anyways.

Now that we had ensured that the currency is separated from the amount, we can add another sub-step to `clean_word`. Here we introduce a trivial replacement and replace all words which are equal to `'€'`, `'eur'`, `'euro'` or `'euros'` with `'€'`.

Next, we added another sub-step to `clean_word` which removes any trailing `',-'` from numbers. We use a regular expression here to also catch cases with

more than a single hyphen. Similarly to the trailing `',-'`, we replace a trailing `'k'` with `'000'`.

The only thing left to do now was to normalise the various decimal formats (i.e. `XXXXX,XX`, `XXXXX.XX`, `XX.XXX,XX`, `XX,XXX.XX`) to `XXXXX.XX` in order to be able parse them in a later step. For the normalisation, we assume that the salary amount has at most two decimal places. We use a regular expression to match 0-2 decimal places with any delimiter symbol and group everything else (i.e. the integer part) together. This gives use the integer part with all delimiter symbols and the decimal part as only numbers. Now we simply remove all delimiters from the integer part and combine both parts again with a `'.'`.

In order to eventually extract the salary, we loop through all cleaned tokens as 2-grams so we can match the currency symbol either before or after the number.

Extracting the Type of Education Another data point we wanted to extract is the level of education required for a given job. As a reference for which types of education exist, we used the website of "The Austrian Education System" [11]. With this information and by subsequently looking at some advertisements we began building some patterns which we could use in our `guess_education` function. Some obvious patterns include `<type>-Abschluss` and `<type>abschluss` with concrete examples being "HAK-Abschluss" and "Pflichtschulabschluss". Since the `guess_education` function receives the input as individual tokens, compound words including acronyms such as "HTL-Abschluss" would already be split into `'htl'` and `'abschluss'`. Given that there are only a handful of acronyms we decided for simplicity's sake to hardcode the most common ones: `'hak'`, `'hbla'`, `'htl'`, `'hlw'` and `'lfs'`. We grouped all of these into the `matura` category. Still, there were some patterns which fall into this category which were not yet recognised, such as "Matura" as a standalone term or "Fachschulabschluss". For these, we simply check if any word contains these terms.

The next two obvious education types were the bachelor's and master's degree. For these, we again simply checked if any word contains the respective term and categorise them accordingly. Another related education type is an FH (Fachhochschule) degree which can be equivalent to either a bachelor's or master's degree. Here we again took the same approach as for the `matura` category, since both "FH-Abschluss" and "Fachhochschulabschluss" are commonly used terms.

With these types for higher levels of education covered, we could guess the education level only for less than half of the results. One type we missed previously was the completion of compulsory school. With compulsory school added to the list, we still were only detecting an education type for around half of all results. After inspecting some of the results which were still missing an education type, we found the two types which were still missing: completed apprenticeship

(often referred to as “abgeschlossene Ausbildung”) and completed apprenticeship (often referred to as “abgeschlossene Lehre”).

With these two types the extraction rate was now at approximately 90%. A few more results could be detected by extracting the words `'studium'` and `'universität'`/`'university'`, since some job advertisements refer to a bachelor’s degree simply as “abgeschlossenes Studium” (completed studies) or list “Universität”/“University” as a requirement.

Extracting the Type of Employment We also wanted to know what type of employment a particular job advertisement pertains to, i.e. whether it is a full-time or part-time position and whether the contract is permanent or temporary. Some websites include this information in their metadata section, for others we would have to extract this data from the body text. Apart from the obvious keywords such as `'full-time'` and `'part-time'`, we also needed to look for the number of work hours per week in the body text, since some advertisements list the work hours explicitly instead of the more vague distinction between full-time and part-time. To extract this, we used a similar pattern to the one for extracting salaries: We loop through all pairs of words. If the first word is a number and the second word is either `'wochenstunden'` or `'hours'`, we parse the number. We assume `'full-time'` if that number is greater than 20, otherwise we assume `'part-time'`.

Extracting the Location The next data point we needed to extract is the location of the job in question. Fortunately, all of the scraped websites include the location as a separate field in their respective metadata section. This way, we could limit our extraction to just this field. This field required a specialised data cleaning step which expands state acronyms. For example, we needed to replace `'W'` with `'Wien'` or `'Oö'` and `'OÖ'` with `'Oberösterreich'`. Additionally, we also replaced the English translations of states and used the German spelling as the canonical version.

Furthermore, we used a database containing all major cities in Austria [12] in order to also catch cases where only the city is specified but the corresponding state is missing. With this database, we can then map the tokenised string to cities and states. In the end, we get a list of all extracted cities and another list of all extracted states. In case we are not able to extract anything from the metadata field, we fall back to running the same extraction function on the whole body text.

Extracting of Certifications Up next was the extraction of certifications. We found an article listing the most prominent certifications of the year 2020 [13]. Incidentally, the authors of this article also retrieved this list from performing a

job board survey. Using the list of certifications mentioned in the article and a manual review of the scraped data, we came up with a list (as can be seen in Table 1) of all the certifications we wanted to extract.

Table 1. List of extracted certifications.

Certification	**Description**
CEH	Certified Ethical Hacker
CISA	Certified Information Security Auditor
CISM	Certified Information Security Manager
CISSP	Certified Information Systems Security Professional
CRISC	Certified in Risk and Information Systems Control
ITIL	Information Technology Infrastructure Library
CIA	Certified Internal Auditor
ISO/IEC 27001	Information Technology Security Techniques
HITRUST	HITRUST Compliance Certification
COBIT	COBIT Certification
ISO/IEC 22301	Business Continuity Management
CGEIT	Certified in the Governance of Enterprise IT
GIAC	GIAC Cybersecurity Certification

In comparison to the other extraction tasks, extracting certifications was straight-forward given this fixed list. There was, however, still one adjustment needed for the data cleaning step with regards to the ISO/IEC certifications since some of them only specified `'ISO'` while others specified `'ISO/IEC'`. A few of them were also missing white space between the acronym and the number, e.g. `'ISO27001'`. For this reason, we added another regular expression replacement to the data cleaning step which normalises all of these occurrences to the canonical `'ISO/IEC #####'` form.

3.3 Data Visualisation

Once the data extraction was implemented, naturally we wanted to visualise the collected data in order to make it easier to reason about and draw conclusions. We chose to use *Jupyter* [14] for this purpose. Specifically we used *Jupyter Notebook*, which is an interactive environment in which you can write code and visualise data at the same time. This makes it very useful when experimenting as you can immediately see the results and you can encapsulate code parts into different sections. This saves a lot of time since you do not need to run the whole project on every change but can test the changes for separate sections individually. *Jupyter* is developed in *Python* [15] but also supports a lot of other programming languages. For our purposes we chose to stick with *Python* since

there are many great tools available with regards to data analysis and visualisation. To further reorganise the collected data we used the well-known *pandas* [16] data analysis and manipulation tool. For data visualisation, we decided to use *Plotly* [17], an open source graphing library. With the power of *Python* and *pandas*, it was relatively straight-forward to convert the collected data into the representation expected by *Plotly* to create graphs. And due to the fact that we were using *Jupyter*, we could immediately see the generated graphs.

4 Findings

Before discussing the findings we made, we will provide a short overview of the amount of data we collected as well as the amount of data which was eventually incorporated to generate the results in this section.

In total, we scraped 1538 job advertisements from all three websites. These are composed of 993 from *Monster*, 282 from *Indeed* and 263 from *StepStone*. These results were aggregated over a time span of about two months by rerunning the scraping tasks roughly once per week in order to scrape newly added job advertisements.

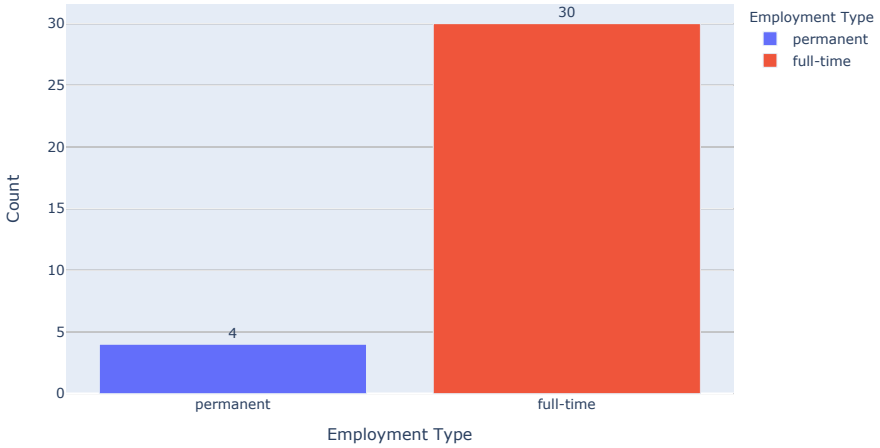

Fig. 2. Employment Type

After a review of the scraped data, we found that many advertised jobs had nothing to do with information security at all and were simply included as a

consequence of the loose search algorithm provided by the job search websites. We therefore decided to add a final filter to the data depending to the job title. Once we added this filter we were left with 121 information security jobs out of 1538 total scraped jobs. All graphs presented in this section are based on this number.

In order to classify the employment type, we used two different distinctions: full-time vs. part-time positions and permanent vs. temporary positions.

In Figure 2 we can see that 95% of cybersecurity jobs are listed as full-time positions and the remaining 5% do not specify this information at all. The permanence of the position is even less common with only 3.3% of permanent positions and the remaining 96.7% leaving this information unspecified. In general, the assumption for leaving this information unspecified is a permanent full-time position, which means that at this point virtually all open cybersecurity positions are permanent full-time positions.

Next we will take a look at the required level of education for information security jobs. As shown in Figure 3, the majority of positions – around 59% – require a bachelor's degree while only around 9% require a master's degree.

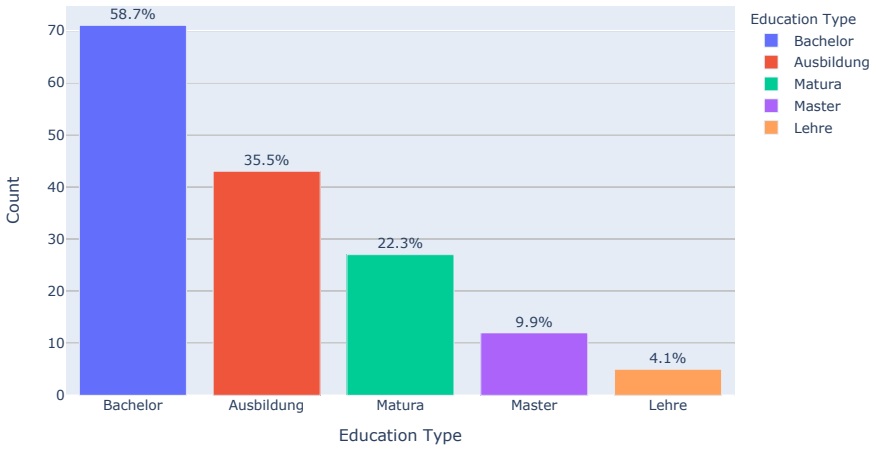

Fig. 3. Education

The second most common requirement is an apprenticeship ("Ausbildung") at around 36%, followed by the "Matura" with 22%. After a manual review of the 4% belonging to apprenticeship ("Lehre"), these seem to be mostly false

positives, at least in our case when only taking information security jobs into account. The false positives in this case are due to the fact the the term “Lehre” is not only used to describe an apprenticeship but also teaching – also referred to as “Lehre” – positions.

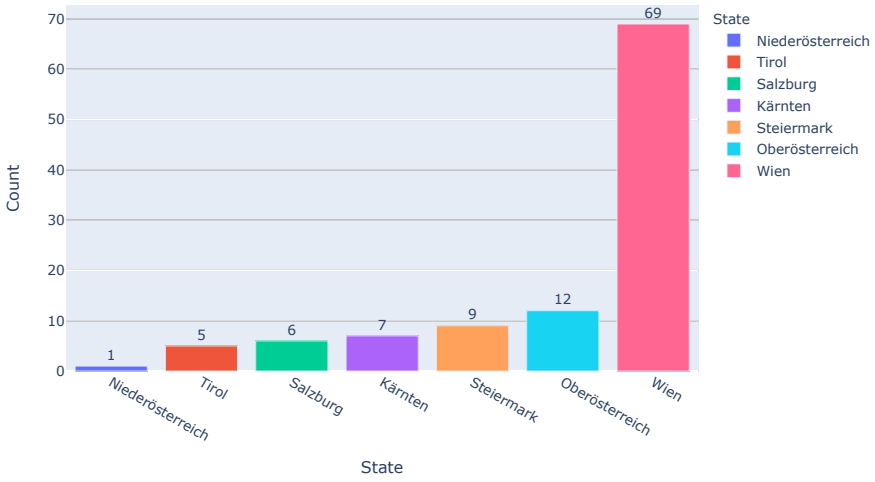

Fig. 4. Location

The distribution across the states of Austria was another point we wanted to analyse. The vast majority of advertisements are for positions in Vienna at about 57%.

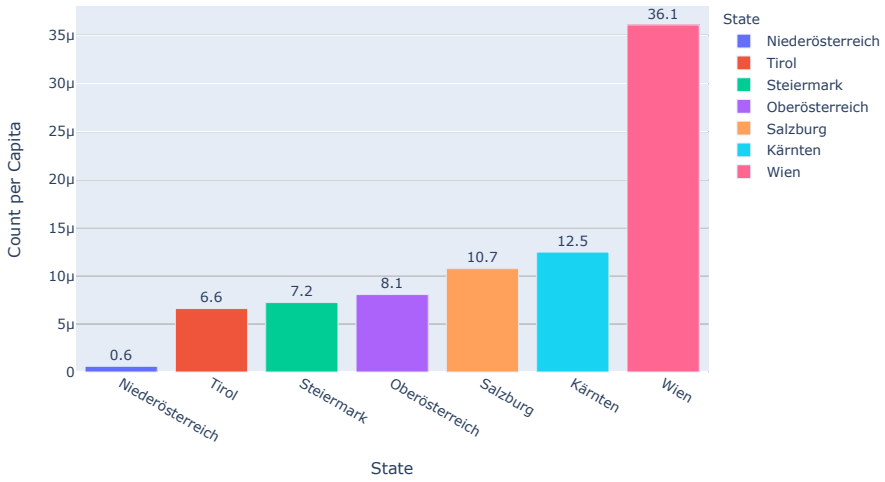

Fig. 5. Location per Capita

Of the other states, only Carinthia, Styria and Upper Austria manage to pass the 5% threshold. Tirol and Salzburg are both at around 4% and Lower Austria is at less than 1% of all advertisements. Vorarlberg and Burgenland do not even have one job advertisement listed.

To further analyse this, we used the population data included in the Austria Cities Database [12] we used for data cleaning and calculated the per capita numbers for each state as seen in Figure 5. The vast majority of advertisements were still for positions in Vienna with almost 3 times as many as the per capita in Carinthia. The relative order of Tirol, Salzburg and Carinthia stayed the same while both Styria and Upper Austria dropped by two places, respectively. Most surprising of all is still Lower Austria with the lowest per capita of all states with more than 0 advertisement. This is especially surprising since Lower Austria encloses Vienna. The close vicinity to Vienna is probably a decisive reason for this dramatic disparity.

Let us now look at the average salary. According to our collected data, the total average salary for information security jobs in Austria is 3715.66 € per month. When looking at the numbers per state, we can see the average salary ranging from 3079 € in Tirol to 4160 € in Salzburg. Given this range we already see a surprisingly huge difference of 1081 € between two the neighbouring states Tirol and Salzburg. The average salary in Vienna is closest to the country's total average with only a 1% deviation. For Lower Austria as well as Burgenland and Vorarlberg we could not extract any salary data.

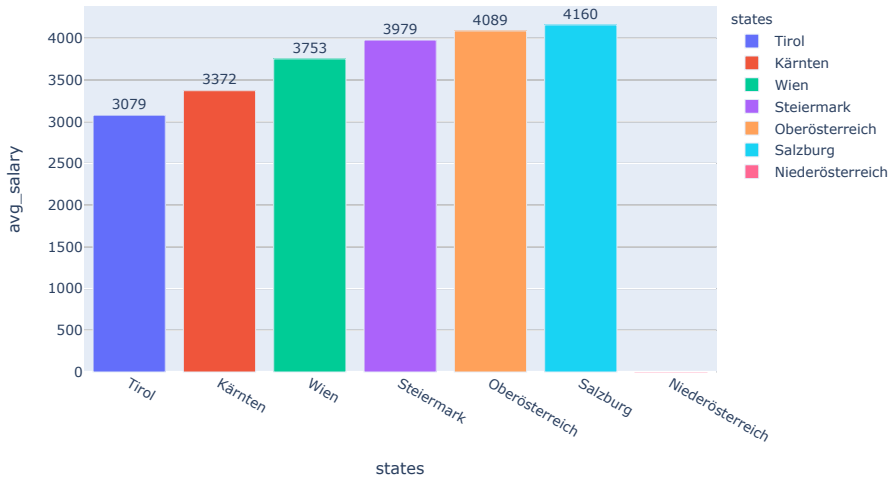

Fig. 6. Salary

Lastly we will analyse the certifications we extracted. Among the most popularly required certifications are CISM, CISA, ITIL, CISSP and ISO/IEC 27001 (refer to Table 1 for the certifications' descriptions) with all being specified in at least 20% of advertisements. With 26% of jobs advertisements mentioning ISO/IEC 27001, it is by far the most requested requirement. In total, 62 out of 121 job advertisements – 51 % –, specify at least one certification as a requirement.

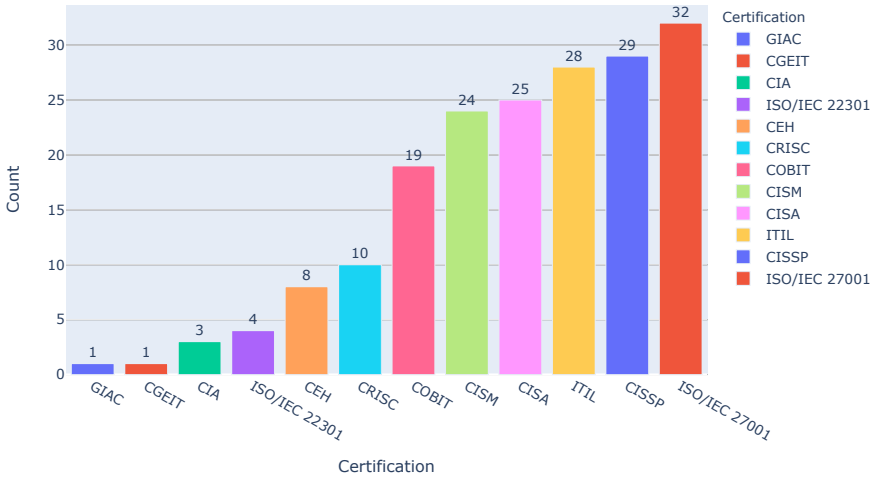

Fig. 7. Certifications

5 Conclusion

In this paper we compared multiple scraping tools highlighting their strong and weak points. Using our chosen scraping tool, we subsequently developed a program for scraping data from three job search sites and used the gathered data to form an overview of the information security job landscape in Austria. Finally, we presented our findings made by analysing the gathered data. These findings can hopefully be used to inform new decisions on what should be taught in universities with regards to information security in order to teach students the skills necessary to succeed in the information security sector.

Some more data points which were not possible to be extracted in this survey could be implemented in a future revision. This includes, for example, the occupational field, which in our survey was not included since only one of the three websites provided this information. Another example includes required knowledge of technologies, which would require a more sophisticated data extraction method than the one used in this survey.

Of course, in order to reduce the chance of outliers skewing the results, the survey could be conducted again within a longer time span and if possible with additional job search websites.

References

[1] Monster Worldwide Austria GmbH. *Monster Homepage*. 2020. URL: https://www.monster.at (visited on 05/24/2020).

[2] Indeed Ireland Operations Limited. *Indeed Homepage*. 2020. URL: https://at.indeed.com (visited on 05/24/2020).

[3] StepStone GmbH. *Stepstone Homepage*. 2020. URL: https://www.stepstone.at (visited on 05/24/2020).

[4] Octopus Data Inc. *Octoparse Homepage*. 2020. URL: https://www.octoparse.com (visited on 05/05/2020).

[5] scraping-bot.io. *Scrapingbot Homepage*. 2020. URL: https://www.scraping-bot.io (visited on 05/05/2020).

[6] Kuaiyi Technology. *ScrapeStorm Homepage*. 2020. URL: https://www.scrapestorm.com (visited on 05/05/2020).

[7] Software Freedom Conservancy. *Selenium Homepage*. 2020. URL: https://www.selenium.dev (visited on 05/05/2020).

[8] Rake Contributors. *Rake Documentation*. 2020. URL: https://ruby.github.io/rake/ (visited on 05/24/2020).

[9] Ruby Contributors. *Ruby Programming Language*. 2020. URL: https://www.ruby-lang.org (visited on 05/24/2020).

[10] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, Inc., 2009. ISBN: 0596516495. URL: https://www.nltk.org.

[11] National Agency Erasmus+ Education. *The Austrian Education System*. 2020. URL: https://www.bildungssystem.at (visited on 05/30/2020).

[12] SimpleMaps.com. *Austria Cities Database*. 2020. URL: https://simplemaps.com/data/at-cities (visited on 05/24/2020).

[13] Ed Tittel, Kim Lindros, and Mary Kyle. *Best InfoSec and Cybersecurity Certifications of 2020*. 2020. URL: https://www.businessnewsdaily.com/10708-information-security-certifications.html (visited on 05/23/2020).

[14] Project Jupyter. *Jupyter Homepage*. 2020. URL: https://jupyter.org (visited on 05/25/2020).

[15] Python Software Foundation. *Python Programming Language*. 2020. URL: https://python.org (visited on 05/25/2020).

[16] Pandas Contributors. *Pandas Homepage*. 2020. URL: https://pandas.pydata.org (visited on 05/25/2020).

[17] Plotly Contributors. *Plotly Homepage*. 2020. URL: https://plotly.com (visited on 05/25/2020).