

Enhancing Cybersecurity: A Machine Learning-Based Framework for Phishing Detection and Mitigation

Anthony Polak 300119082, Ibrahim Maalej 300145176, Emanuel Ruiz 6976748

apola087@uottawa.ca, imaal081@uottawa.ca, eruiz053@uottawa.ca

Department of Computer Science, University of Ottawa

Project Supervisor: Mohamad Hoda

April 24, 2024

Abstract	3
Introduction.....	3
Literature Review	4
Phishing Detection Using Machine Learning Algorithms	4
Datasets	5
Web Infrastructure	5
Proposed Framework	6
Methodology.....	6
Dataset	6
Methodology.....	6
Dataset Shape	7
Dataset Attributes	7
Parsed URL String Features	7
Domain Query Features.....	8
Networking Features	8
Security Features	8
Model.....	9
Technology	9
Dataset	9
Model Selection	9
Feature Selection and Analysis	12
Training Process	17
Final Implementation and Evaluation.....	19
Web Infrastructure	19
Creating the Virtual Machines	20
Domain Name Set Up	21
Shared Configurations between the Servers	23
Webpage Design	25
API Setup.....	29

Successes	31
Section Summary	32
Results and Discussion	33
Next Steps	33
Conclusion	34
References	34

Abstract

Abstract---Phishing attacks remain a persistent and pressing concern in today's emerging cyber security issues, garnering considerable attention in the community due to their pervasive nature. While various methods exist for detecting phishing URLs, the challenges of identifying these malicious URLs persist as attackers continually adapt their tactics to evade developed detection mechanisms. Our project introduces a supervised-learning-based model for detecting malicious URLs with significant accuracy and efficiency, which is integrated into a user-friendly website tool that empowers individuals to make informed decisions when encountering suspicious URLs online. By providing a reliable and easy to use phishing detection tool to everyone, our project aims to enhance cybersecurity resilience, awareness, and mitigate the risk of falling victim to a phishing attack in the ever-changing digital landscape.

Introduction

Machine learning is a branch of artificial intelligence focused on enabling computers to learn from data, that aims to automate the decision and prediction making process based on patterns in data.

Phishing is a type of cyber-attack where attackers attempt to deceive users into providing sensitive information through numerous different methods, primarily for some sort of financial or information gain.

The real-world impact and damage caused by phishing attempts is well documented. The FBI's 2021 Internet Crime Report showcases that 22% of all data breaches were due to phishing attacks. This becomes a serious issue as the absence of a layer of protection between the user and URLs online means users will have to exercise due diligence to tell URLs apart. However, an Intel study shows that 97% of people fail to identify phishing URLs from legitimate ones.

The overarching problem addressed by this project is the persistent vulnerability of individuals and organizations to phishing threats.

We set out to design a sophisticated AI model-based tool aimed at enhancing cybersecurity and protecting users through the detection of link-related phishing attacks in web content. The goal was to train a machine learning model to predict whether a URL was part of a phishing attempt or not. A dataset was built to train the model, an algorithm was selected to make accurate predictions, and a web interface was built to facilitate user interaction with the model.

The project was an experience that allowed our team to produce something practical using trending technologies and research. It allowed our team to gain a lot of new knowledge and skills in high demand through productive action. Our team was also able to use knowledge which was previously gained through our Computer Science studies.

Literature Review

Phishing Detection Using Machine Learning Algorithms

Traditional methods for phishing detection primarily rely on rule-based systems and signature-based detection. Rule-based systems involve creating a set of rules to identify phishing emails based on known characteristics such as suspicious URLs, misspellings, or mismatched sender information. Signature-based detection compares incoming emails to a database of known phishing emails or patterns.

Advancements in phishing detection methodologies have been driven largely by the integration of machine learning (ML) techniques. ML algorithms, particularly supervised learning models like Support Vector Machines (SVM), Random Forest, and Neural Networks, have shown promise in detecting phishing attacks by analyzing large datasets of potential features.

- Mahajan, Rishikesh, and Irfan Siddavatam. "Phishing Website Detection using Machine Learning Algorithms." *Research Gate*, vol. 181, no. 23, Oct. 2018, <https://doi.org/10.5120/ijca2018918026>.
- Gana, Noah N., and Shafi'I M. Abdulhamid. "Machine Learning Classification Algorithms for Phishing Detection: A Comparative Appraisal and Analysis." *Research Gate*, Jan. 2020, <https://doi.org/10.1109/NigeriaComputConf45974.2019.8949632>.

The above were a few examples of papers chosen amongst many others reviewed and provide comprehensive studies on phishing detection using machine learning techniques. They review various machine learning algorithms, including SVM, Random Forest, Neural Networks, etc, for identifying phishing attacks. They evaluate and compare the performances of each model against each other, analyzing their strengths and limitations.

Datasets

Along with data pre-processing techniques and standards, two studies are referred to which provide starter datasets that we can merge and transpose to utilize as a framework to build on top of.

- <https://doi.org/10.1016/j.dib.2020.106438> by Grega Vrbančič
 - 111 attributes, 58,645 records
- <https://doi.org/10.17632/c2gw7fy2j4.3> by Abdelhakim Hannousse
 - 87 attributes, 11,430 records

The studies encompass a diverse array of approaches and methodology aimed at detecting phishing links utilizing machine learning techniques, each of which introducing novel feature sets to capture the characteristics of phishing URLs.

Web Infrastructure

Creating servers that offer a service requires the use of several existing technologies. Many guides and other forms of documentation were consulted to install and configure these technologies to provide the custom services that our applications required and were intended to offer.

For domain name registration, we consulted the guides offered by Digital Ocean and our registrar Namecheap.

- <https://docs.digitalocean.com/products/networking/dns/getting-started/dns-registrars/>
- <https://www.namecheap.com/support/knowledgebase/article.aspx/9776/2237/how-to-create-a-subdomain-for-my-domain/>
- <https://docs.vultr.com/introduction-to-vultr-dns>

To create certificates to provide HTTPS connections to our servers we used the Certbot guide:

- <https://certbot.eff.org/instructions?ws=apache&os=centosrhel7>

To create webpages, we followed various tutorials and consulted documentation such as the docs provided by the Mozilla Development Network.

- <https://developer.mozilla.org/en-US/docs/Web>
- <https://getbootstrap.com/docs/5.2/utilities/spacing/>

- https://www.w3schools.com/js/js_html_dom_nodes.asp

To configure Flask, we followed the Flask project documentation:

- <https://flask.palletsprojects.com/en/2.3.x/patterns/urlprocessors/>
- <https://flask-cors.readthedocs.io/en/latest/>
- <https://auth0.com/blog/developing-restful-apis-with-python-and-flask/>

Proposed Framework

Methodology

Our methodology to implement our solution is as follows: Research the problem to understand the intricacies of phishing attacks and machine learning models, find the data containing diverse and up to date benign and phishing links, utilize machine learning algorithms to build a model trained on this dataset that's able to accurately classify phishing links, and then user-friendly tool using this model to assist users in staying secure online.



Fig 1. Our Methodology

Dataset

Methodology

Our methodology begins with the acquisition of a base set of data utilizing the datasets provided by the literature above. The datasets are then enriched and transposed to match each other for merger utilizing Python's Pandas library in a Jupyter Notebook. The data is then cleaned, where outliers are identified and missing values are handled, as well as addressing other inconsistencies in the merged data. Features are then engineered to

provide further insight into the data provided to continuously improve the quality of the training data.

Dataset Shape

The resulting merged dataset following thorough edits utilizing Pandas yields us with 40,000 records categorized as either phishing or non-phishing, in which 34.64% of total records are phishing URLs, and 65.36% non-phishing URLs. Each record contains 116 attributes which are selected based on the insight they provide into the characteristics of a URL. These attributes encapsulate various aspects such as URL structure, domain information, textual content, which will then enable the classification model to discern between legitimate and malicious URLs effectively. The next section provides an overview over the attributes within the dataset.

Dataset Attributes



Fig 2. URL Composition

Typical URLs can be dissected into different subcomponents that allows us to take a detailed approach to filling the dataset with useful features. This can range from metadata that can be extracted from the string subcomponents directly (HTTPs as the protocol? Redirections present?), or utilizing the subcomponents to query more information, such as the domain providing DNS records through Python's who.is API. Some of the features included are as follows:

Parsed URL String Features

Once the URL is parsed, we check for the length of the URL and certain sections such as the domain. Most of the features present in our dataset can be categorized as the quantity of certain special characters under different parts of the URL. After parsing the URL, each useful section's quantity of special characters `{!, @, #, $, %, ^, &, *}` is stored as separate features in the dataset `{qty_dot_url, qty_hyphen_url, ..., qty_dollar_file}`.

Furthermore, the ratio between special and non-special characters is calculated to provide more insight.

Further information can be extracted by checking for certain details in the URL besides information about the string, such as the presence of certain suspicious keywords {Bank, Login, Verification, Invoice, Etc}. We can also check the presence of HTTPs as the protocol, or if other sections contain the sub-string HTTPs to trick users. We can also check for known URL shortening services that hide URLs, such as TinyURL, and Bitly.

Domain Query Features

The domain provides us with access to query certain information that expands the insight we could derive from the URL, besides string features. Utilizing Python's who.is API, we can pull the DNS record related to the domain (if available). This provides us with features such as the time the domain was activated, and when it will expire, as well as the frequency the record was updated with respect to time.

Furthermore, we are also able to query some of the more useful numerical features such as the ranking of the domain amongst the top 1 million as recorded by Tranco, or whether the domain is indexed within google to begin with, in which non-indexed domains and unranked pages are considered suspicious.

Networking Features

Utilizing the Python's Socket library we extracted features about the low-level networking information. Firstly, the response time taken while querying the domain through the Socket library is recorded, and the associated IP address is stored to query more information, and if many IP addresses are resolved, it gets stored under the `qty_ips_resolved` feature. The IP address allows us to query information such as the presence of Autonomous System Numbers (ASNs).

Security Features

Certain security and safety features are enumerated based on information extracted from the URL and domain, such as the TLS/SSL certificate validity, in which a valid certificate ensures secure communication, such that an invalid certificate highlights a potential risk.

Model

Technology

Based on our research and testing of existing solutions and implementations of phishing URL detection, our group sought to implement a supervised learning solution utilizing the powerful random forest model and its ensemble learning technique. When training the model, we decided to focus on physical features of URLs such as length with some signature-based ones such as google page rank mixed in, choosing both categorical and numerical input variables that complement the random forest classifier. In our implementation of this solution, we chose Python and the scikit-learn library as our main building blocks as they are tried-and-true tools perfect for our realizing our idea.

Dataset

As previously covered in the “datasets” section, the final chosen dataset we trained our model on contains 40,000 instances (rows) with 116 features (columns) each, with a distribution of 65.35% benign (non-phishing) and 34.64% malicious URLs. The wide variety of features, large number of instances, and distribution percentage of URLs were continuously tested and updated to ensure the best possible results until we settled on our final parameters.

Model Selection

To ensure the best possible model was chosen for the task, we pitted many potential models against each other and compared things like:

Precision (out of all the links your model labeled as phishing, how many were phishing links) and Recall (out of all the actual phishing links out there how many did our model correctly identify as phishing):

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \quad \& \quad Precision = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Fig 3. Recall & Precision Formula

F-1 score (the balance between recall and precision that gives you an overall sense in how well our model performed in detecting phishing links):

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Fig 4. F1 score Formula

Test and train scores and accuracy:

$$Accuracy = \frac{True\ Negative + True\ Positive}{True\ Negative + False\ Negative + True\ Positive + False\ Positive}$$

Fig 5. Accuracy Formula

We also utilized tools such as cross-fold validation and the Matthew's Correlation Coefficient (MCC) to make sure no model was overfitting and features were properly correlated.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

Fig 6. Matthew's Correlation Coefficient Formula

Taking these results, we used tools such as confusion matrices and libraries like sealion to graph and display them for further analysis.

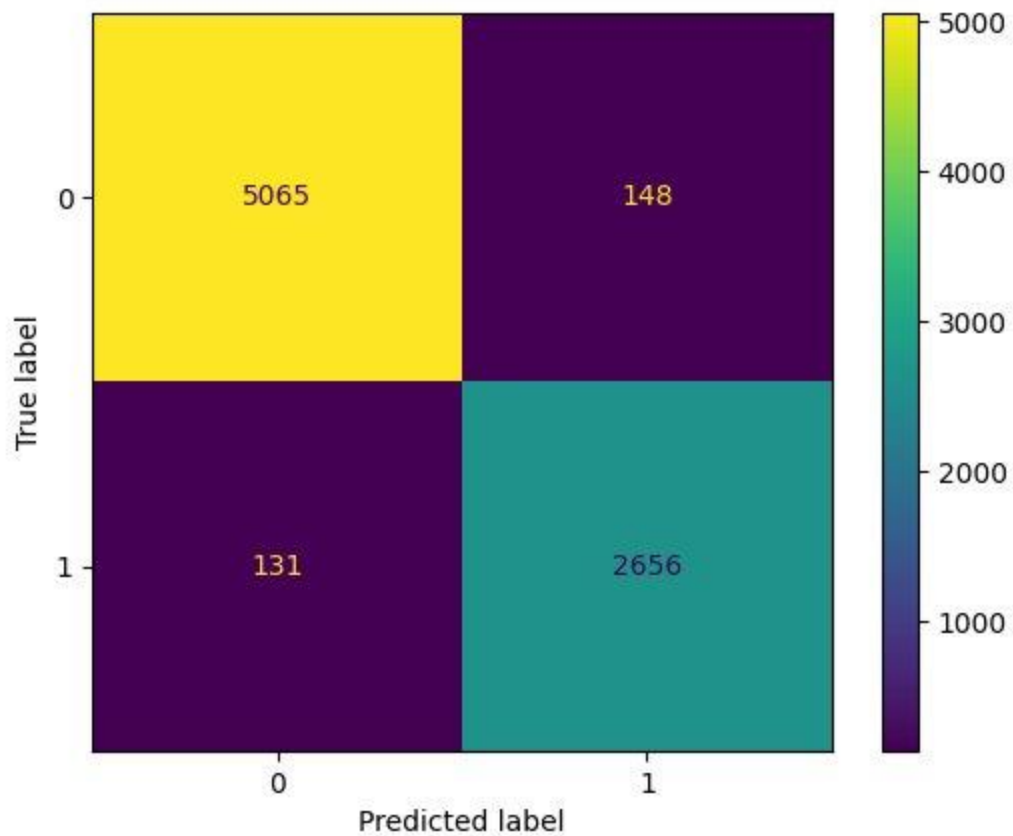


Fig 7. Random Forest Confusion Matrix

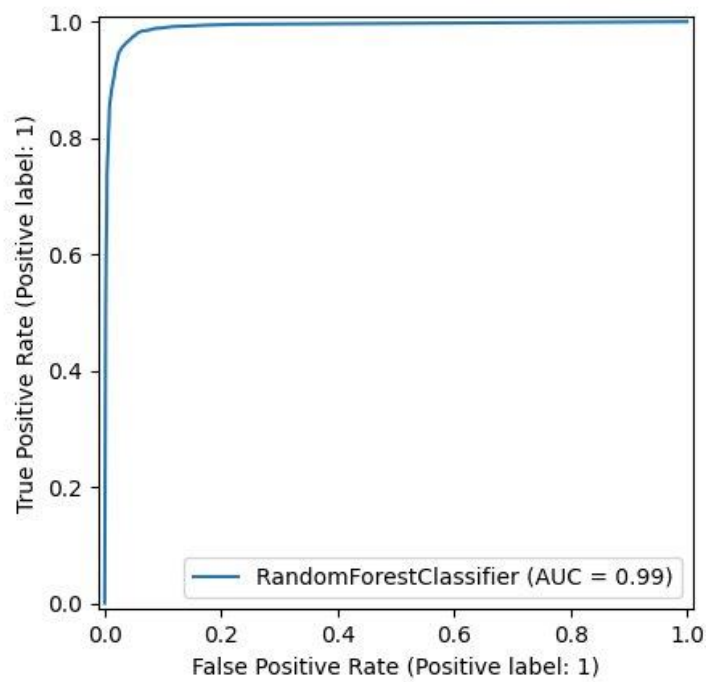


Fig 8. Random Forest AUC/ROC Display

In the end the Random Forest classifier came out on top against 6 other models (Light GBM, Logistic Regression, Decision Tree, Naive Bayes, Support Vector Machine, Multilayer Perceptron) in almost all metrics with a 96% score in each metric as well as a 91% MCC and 96.1% test and 99% train scores. Based on the results it was a very close call with the light GBM classifier, but we ended up choosing random forest in the end due to its simplicity and robustness, as well as familiarity.

MODEL SELECTION					
	PRECISION	RECALL	F1-SCORE	TEST SCORE	TRAIN SCORE
Random Forest	0 0.97 1 0.94	0 0.97 1 0.95	0 0.97 1 0.94	0.961	0.999
Light GBM	0 0.97 1 0.94	0 0.97 1 0.95	0 0.97 1 0.94	0.960	0.977
Logistic Regression	0 0.95 1 0.87	0 0.93 1 0.91	0 0.94 1 0.89	0.922	0.924
Decision Tree	0 0.95 1 0.92	0 0.96 1 0.91	0 0.96 1 0.92	0.944	1
Naive Bayes	0 0.92 1 0.80	0 0.89 1 0.86	0 0.91 1 0.83	0.879	0.879
Support Vector Machine	0 0.70 1 0.81	0 0.98 1 0.20	0 0.81 1 0.31	0.707	0.707
Multilayer Perceptron	0 0.83 1 0.95	0 0.98 1 0.63	0 0.90 1 0.75	0.859	0.862
BEST	Random Forest	Random Forest	Random Forest	Random Forest	Decision Tree

Fig 9. Model Tournament Selection Results

Feature Selection and Analysis

When deciding which features to choose, we chose to keep the 112 features from one of our very solid datasets and simply apply them to the other as well as adding 4 new ones, as they cast a wide cover over what we were looking to measure for. To implement and extract these features we had to split up and preprocess each URL (as seen in the datasets section) into 6 parts, the URL, domain, directory, file, params and record, using URLparser.

```

def featureExtraction(url):
    print('in featureExtraction')
    if validateURL(url) == 0:
        return "Not a valid URL"
    parsed = url_parser(url)
    print("this is parsed:", parsed)
    domain = parsed.netloc
    print("this is domain:", domain)
    directory = parsed.path.rsplit(sep: "/", maxsplit: 1)[0]
    print("this is directory:", directory)
    file = parsed.path.rsplit(sep: "/", maxsplit: 1)[-1]
    print("this is file:", file)
    params = parsed.query
    print("this is parameters:", params)
    try:
        record = whois.whois(domain)
    except:
        record = None
    print("this is dir(record): ", dir(record))
    print("this is record:", record)

```

Fig 10. Feature Extraction

Once we had gotten through that, validating that our chosen features were solid and how they worked at multiple levels of granularity was the next step of our implementation. We used many feature importance tools such as Variance Inflation Factor (ViF), LIME, SHAP, and Permutation Importance which can have their outputs seen below. (can't decide whether photo first and then discussion, or definition, photo, then discussion)

Variance Inflation Factor (VIF)

VIF allows us to assess multicollinearity among features to ensure that each feature provides unique information to the model.

59	qty_hyphen_file	6.389052
103	qty_nameservers	5.676059
75	file_length	4.568873
100	time_domain_activation	3.396401
102	qty_ip_resolved	3.006201
106	tls_ssl_certificate	2.424170
104	qty_mx_servers	2.304309
109	domain_google_index	1.661424
108	url_google_index	1.658722
101	time_domain_expiration	1.648173
99	asn_ip	1.516875
107	qty_redirects	1.422552
105	ttl_hostname	1.374359
97	time_response	1.338245
38	domain_in_ip	1.155511
110	url_shortened	1.128012
39	server_client_domain	1.028662
98	domain_spf	1.022042

Fig 11. VIF Inflation Factor

By looking at the above chart generated by VIF, we can see that the selection of some of our chosen features show good scores (typically VIF around 5 is ideal, but it's a tricky measurement that only provides an estimation). These good scores, along with many repeating prominent feature seen in future analysis, show good confidence that our chosen features are important.

LIME

Lime provides interpretable insights into the model's predictions for specific instances by approximating the model with a simpler, interpretable model.

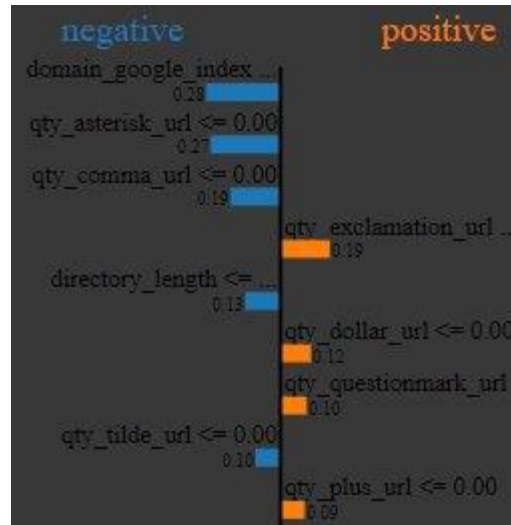


Fig 12. LIME Feature Insights

The above insights display the impact of each feature in our model on a more local grain by approximating the model with a simpler, interpretable model. We can see that for this particular prediction, a negative value of domain_google_index and a positive value of qty_exclamation_url played a large role in the end classification. Showing us that while some features such as ones involving qty may not seem to have an impact at a higher level of granularity, when we go to a lower level they do seem to play an impact in some individual predictions, reinforcing our choices.

SHAP

SHAP gives us values to understand each feature's overall contribution to the model's predictions and their impact on classification outcomes.

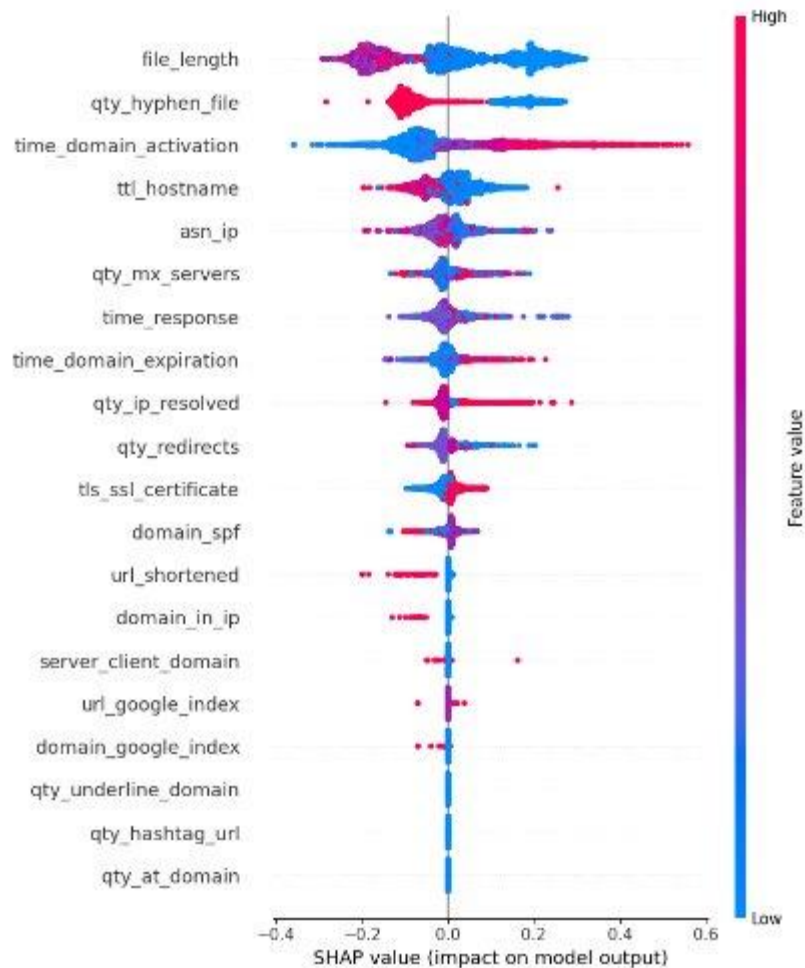


Fig 13. SHAP Summary Graph

The above summary graph displays the impact of some of our features on overall model classification (couldn't fit all due to size reasons of course). From looking at the graph for example, we can see that a high feature value of `time_domain_activation` leads to positive SHAP values (or has a positive impact on model output) while a lower value leads to negative SHAP values, which intuitively makes sense as logically speaking phishing URLs or sites would have much newer and sporadic domain activation times than trustworthy long-standing ones.

Permutation Importance

Permutation measures the importance of each feature by observing the change in model performance when the feature's values are randomly permuted.

directory_length	0.180 +/- 0.003
time_domain_activation	0.062 +/- 0.002
length_url	0.016 +/- 0.001
qty_dot_domain	0.010 +/- 0.001
ttl_hostname	0.007 +/- 0.001
qty_nameservers	0.006 +/- 0.001
asn_ip	0.006 +/- 0.001
time_response	0.005 +/- 0.001
qty_mx_servers	0.003 +/- 0.000
qty_ip_resolved	0.003 +/- 0.000
time_domain_expiration	0.003 +/- 0.001
domain_spf	0.002 +/- 0.000
qty_slash_url	0.002 +/- 0.000
qty_hyphen_url	0.001 +/- 0.000
qty_dot_url	0.001 +/- 0.000
qty_hyphen_directory	0.001 +/- 0.000
qty_underline_file	0.001 +/- 0.000
domain_in_ip	0.000 +/- 0.000
qty_comma_file	0.000 +/- 0.000
qty_hyphen_file	0.000 +/- 0.000

Fig 14. Permutation Importance

The above table displays the impact of each feature in our model when feature's values are randomly permuted (with their variation +/- on the right). From looking at the table we can see it shares many features with our SHAP summary graph, and that some major important features include directory_length and once again, time_domain_activation, amongst others.

The above feature analysis shows us that our best features chosen have an impact on the model's performance and are somewhat consistent across many granularities, ensuring our feature importance.

Training Process

In our initial training phase, we employed the random forest model with base parameters and a train-test-split of 75:25 on the untrimmed merged dataset to establish a baseline of 93% overall accuracy for our model's performance. We then meticulously adjusted hyperparameters, k-fold cross validated, added and removed features, and experimented with dataset size and distribution throughout the training process, until we reached a notable enhancement in overall accuracy from 93% to 96%.

Some major hurdles we faced when conducting iterations were overfitting due to dataset size, underfitting due to loss of features, and time constraints placed by limited access to google and other api querying services.

While our initial dataset of 130,000 entries seemed to provide good numerical results with the model, we soon learned during real testing on randomly selected links from Phishtank that the high percentages in training, test, and the other measurement scores were due to overfitting on the dataset. Since it was so large and with so many features, the powerful model had simply learned the distribution of the data and its underlying patterns. This led to good numerical training scores but very poor generalization and test scores when quizzed on non-familiar links. Due to this, we had to experiment with changing the dataset size to prevent overfitting.

By using the feature importance tools mentioned above I attempted to single out and remove seemingly unrelated features contained in our dataset, but through testing it appears that these unassuming features were correlated with other strong features, or together in a way that the analysis techniques couldn't display. This was seen when we attempted to slim down the feature set to 66 features (removing a lot of the low correlation "qty" features and such) which resulted in heavy underfitting as the features and data wasn't descriptive enough for the model to capture and learn an underlying trend. This led us to keeping all original features and simply adding some more of our own features like "ratio_special_chars" instead to enhance the data.

Another hurdle, and frankly the largest, was the limitations on querying systems such as google's API and the errors they produced. This led to us frequently getting blocked and rate limited when trying to make changes and testing code, as well as bringing up many errors that had to be handled and muddling our results as sometimes we would make a prediction on a URL that would be correct, but then predict it again a second later with a different incorrect result due to the model being blocked from querying for the info it needs to conduct our feature extraction (this appeared frequently when we transferred the model to the webserver, as we think google is specifically blocking it due to it not being a host machine). Ideally, we would've liked to do much more testing with new features, less features, etc, finding the exact thresholds for greater results, but this had severely limited us and caused multiple day delays between attempted experiments.

Throughout all this testing we also made sure to validate our model with 4-10 k-fold cross validation to further prevent overfitting and conducted manual hyperparameter tuning.

In our final iteration we ended up with a trimmed dataset of 40,000 entries and 116 features (115 and 1 target), a train-test-split distribution of 80:20, had done multiple cross-validation tests of sizes 4-10, and settled on the hyperparameters of (oob_score=True, n_jobs= -1, random_state = 42, max_features = 200, n_estimators = 60).

Final Implementation and Evaluation

In our final implementation we had over 900 lines of python code including packages such as scikit-learn, pandas, etc, trained on a final processed quality dataset of 40,000 entries, had 116 features (115 features and 1 target), had manually tuned the hyperparameters for best results, validated using 4-10 fold cross validation and a 91% MCC, and had made a strong, reliable and efficient model for detection with 96% accuracy on average.

Taking this final model, we dumped it into a .joblib file, downloaded and transferred the saved version to the webserver for future implementation and use.

Web Infrastructure

The goal of this section was to provide an accessible and secure graphical interface to interact with the trained machine learning model. The graphical interface was also intended to serve as a visual tool for presenting the project. The ability for other users to interact with the model in an accessible way assisted in making the project feel more complete and more practical, as opposed to having something that existed on paper only or involved performing the same set up as our team.

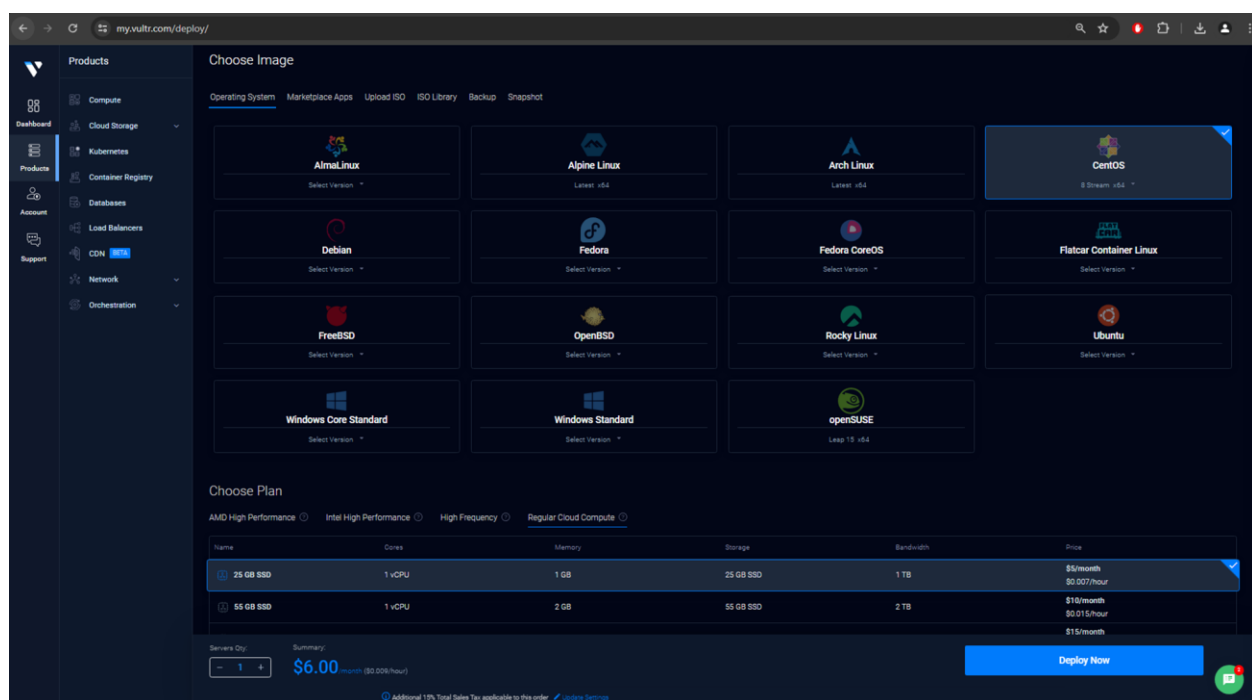
The initial idea involved creating a browser extension that a user could simultaneously use while reading their e-mails through the web browser. The extension was to act as the front end of this application. It would interact with a backend server which offered processing through API endpoints. Some of those endpoints were to involve the trained model by receiving URL as text data and using the model to return a prediction.

As the project progressed, the idea of the extension was put on hold and a website was created. The website offered most of the desired features that were intended to be had with the extension. The ability to have predictions be done within the same window as a webmail interface was not present, yet some new benefits were such as having a larger space to present information to our users. The short-term pivot from extension to website mainly involved feasibility. This project was an academic endeavor intended to provide our team with many learning opportunities. With little to no experience in web infrastructure, a lot of learning was done involving web development. This included going through tutorials on making simple responsive websites with HTML, CSS, and JavaScript. Having gained enough skills to create a website the pivot occurred to be able to produce a Minimum Viable Product of our project concept.

Below we discuss more technical details on how the servers were created and configured. We also include additional information such as some learning resources that were used and some of the design journey.

Creating the Virtual Machines

To be able to offer information processing services as our prediction project intended, we needed hardware resources. These hardware resources were to be used for computations, had to be configurable and had to be able to be opened to the public (through internet ports). To provide us these resources, we used Vultr which is a cloud computing platform where we could host virtual machines.



Above is a screenshot of the Vultr web interface where one can see that a variety of hardware resources are offered for a virtual machine, with varying prices relative to the performance. One can also choose the operating system that the virtual machine will host.

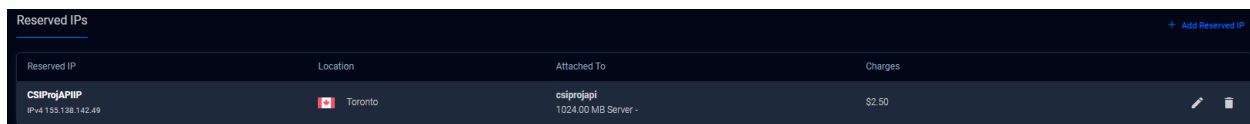
For our project, we deployed two virtual machines. Initially only the API server was deployed. The hardware resources were as follows: “AMD High Performance” Plan - 1 vCPU, 1024 MB RAM, 25 GB NVMe, 2.00 TB Transfer. This plan cost approximately \$6 per month. Paying for automatic backups was not a selected choice but progress was saved through manual snapshots at smaller fees.


Later, the Web server was deployed with similar properties: “Regular Cloud Compute” - 1 vCPU, 1024 MB RAM, 25 GB SSD, 1.00 TB Transfer. This plan cost \$5 and the performance

was also enough for our purposes. We had the option to upgrade our plan to provide us with larger computing resources if at any point during our development and testing it was believed that we did not have enough computational power.

The operating system that was chosen for this VMs was Centos Stream 9. This is the latest version of the distribution. It is a Linux distribution which is similar to Fedora and Red Hat Enterprise Linux. As such, it provided our group with a familiar environment to install software through packages, and edit code and configuration files. It carried no additional costs to install unlike the Windows OS options.

Both Virtual machines were chosen to be hosted in the Toronto, Canada datacenter. This would keep our data in Canada. Additionally, having our virtual machines in a closer datacenter would mean we would have faster response times when retrieving and sending data between the client devices and the servers.

A screenshot of the Vultr web interface showing a table of reserved IP addresses. The table has four columns: Reserved IP, Location, Attached To, and Charges. There is one row of data. The 'Reserved IP' column shows 'CSIPProjAPIIP' and 'IPv4 155.138.142.49'. The 'Location' column shows a Canadian flag and 'Toronto'. The 'Attached To' column shows 'csipprojapi' and '1024.00 MB Server -'. The 'Charges' column shows '\$2.50'. There are edit and delete icons at the end of the row. A '+ Add Reserved IP' button is in the top right corner.

Reserved IP	Location	Attached To	Charges
CSIPProjAPIIP IPv4 155.138.142.49	 Toronto	csipprojapi 1024.00 MB Server -	\$2.50

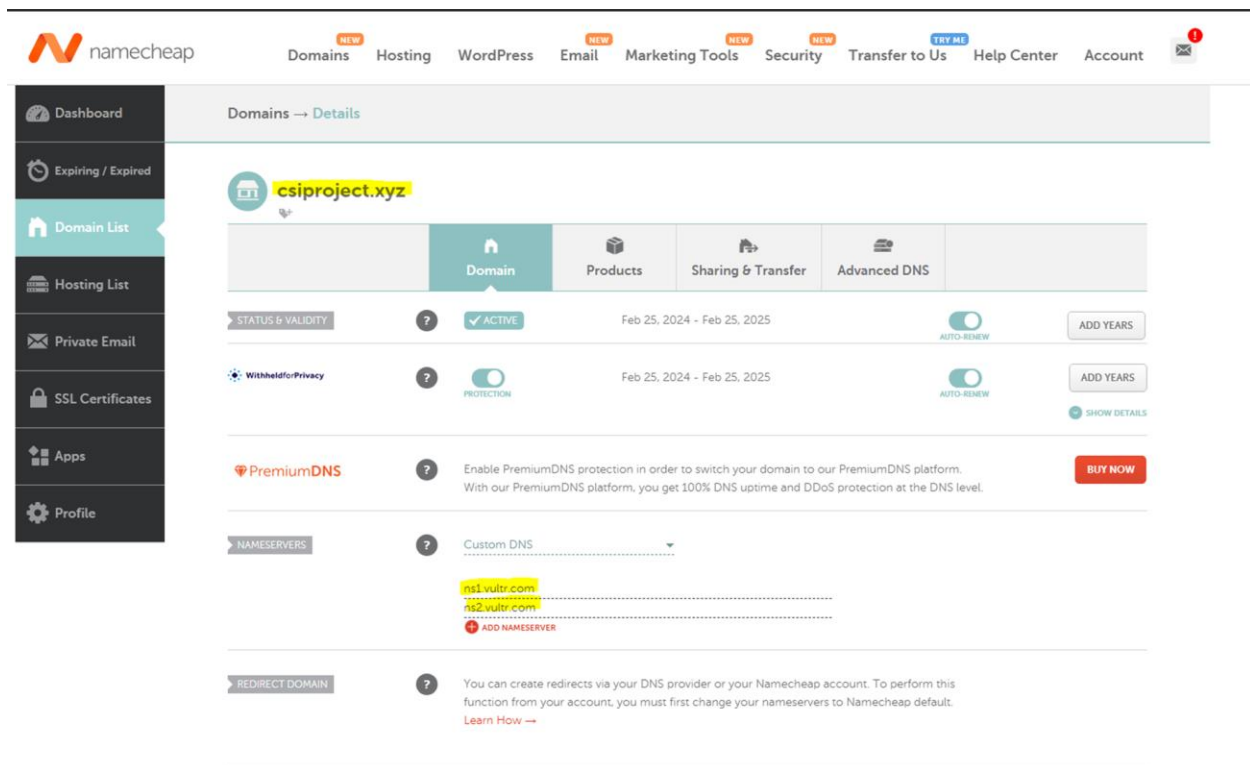
Vultr allowed us to reserve IP addresses for a cost. Two IP addresses were reserved such that if a server needed to be redeployed, the other would not need to be modified to accept a new IP address. Also, DNS would require less configurations as well once our team moved on to seeing up a domain name for a more accessible use.

Our API server used the 155.138.142.49 IPv4 address. The Web server used the 155.138.131.60 IPv4 address. No special configurations were made for IPv6.

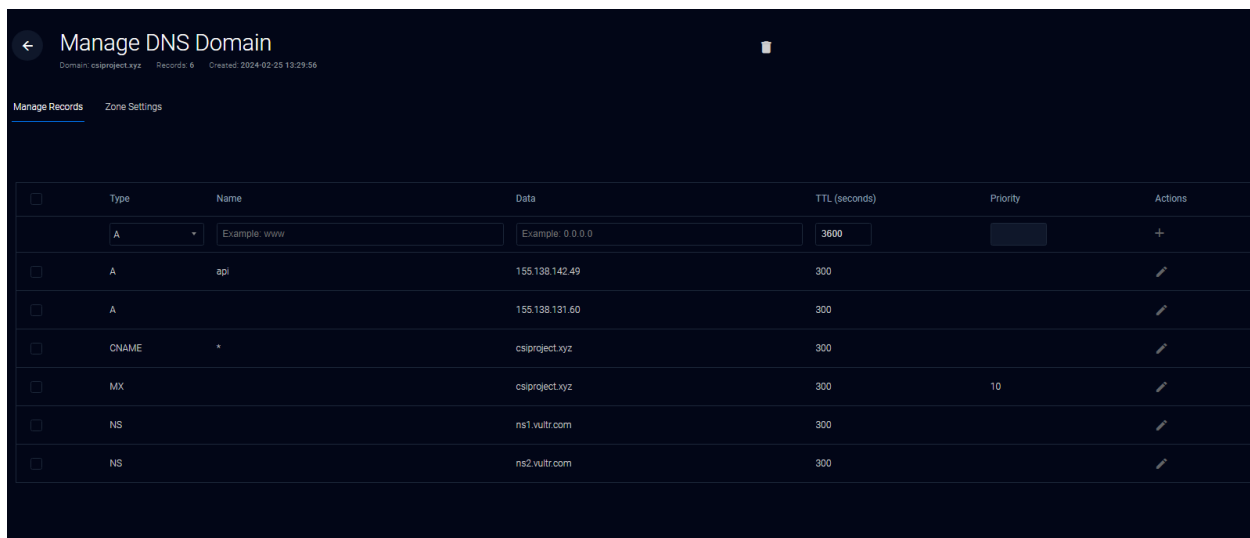
Vultr offers a network firewall service. This was configured to ensure the relevant ports were opened for traffic to our virtual machines. Port 22 was opened to be able to SSH to our servers using MobaXterm. Port 80 and Port 443 were opened to allow HTTP and HTTPS traffic, respectively.

Domain Name Set Up

To make the website more presentable, a domain name was purchased from Namecheap. The domain is csiproject.xyz . The xyz Top-Level Domain name was chosen due to its accessible cost.



To associate our domain name with the IP addresses of our servers, we used public DNS which had to be configured with our registrar Namecheap and our cloud-compute provider Vultr. From the Namecheap console we submitted the hostnames of the Vultr DNS servers.



Using the Vultr DNS console, we added a DNS domain item which was configured as above. Adding our IP addresses as A record as shown above, allowed our web servers to be more accessible as the names are easier to remember than IP addresses. The Web Server

was addressable via the csiproject.xyz URL. The API server became addressable through api.csiproject.xyz.

It was interesting to learn how easily subdomains could be added using these DNS services once our team owned the second-level domain. The DigitalOcean guide was used to configure DNS from the Namecheap console. The Vultr guide was used to set up the subdomain in the Vultr console.

Shared Configurations between the Servers

Besides the configurations done in Vultr, the servers shared some configurations which were done interacting directly with the OS using MobaXTerm as an SSH client.

Through initial troubleshooting, it was learned that Centos Stream 9 has firewall rules up by default. Our team had to allow traffic to the necessary ports using bash – a command language used on the Linux operating systems to interact with the OS.

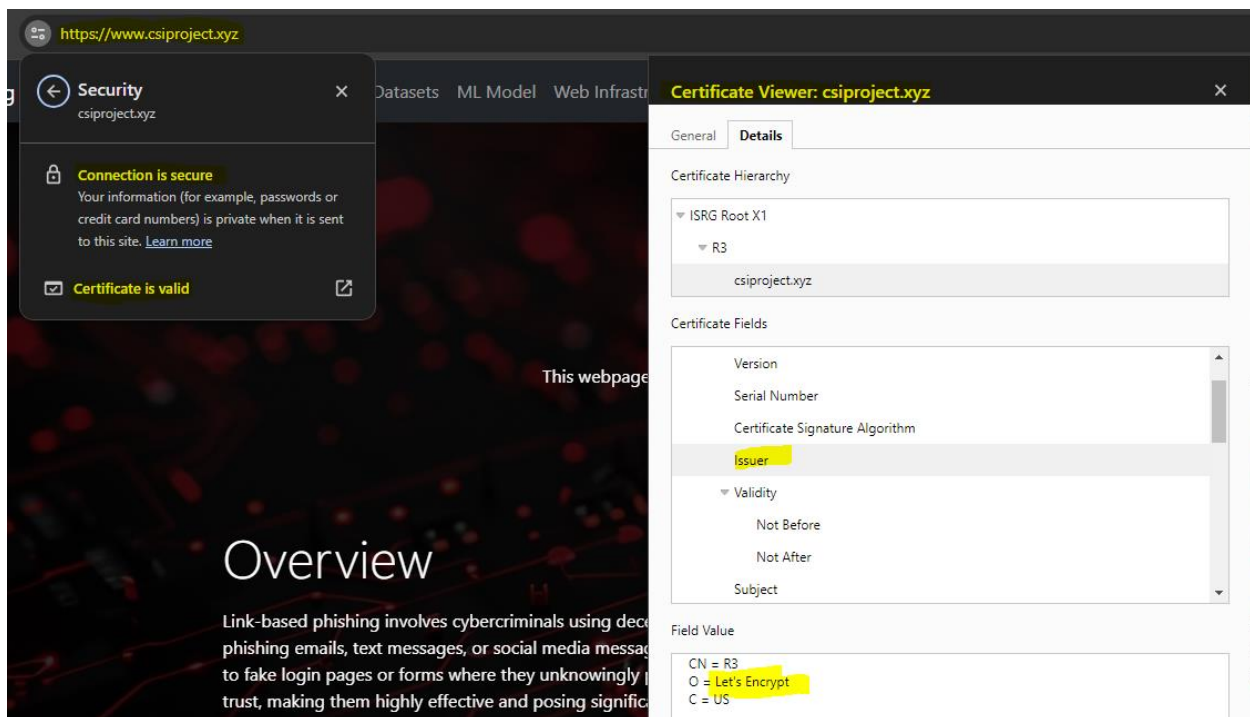
```
[root@csiprojwebserver ~]# firewall-cmd --state
running
[root@csiprojwebserver ~]# firewall-cmd --list-all
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: enp1s0
  sources:
  services: cockpit dhcpv6-client http https ssh
  ports: 80/tcp
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
[root@csiprojwebserver ~]#
```

Our servers have OS-level firewalls active which allow traffic for the services ssh, https and https. These were enabled on the “public” firewall zone. This configuration was made permanent so that it is maintained after a reboot.

Secure communication with the servers was configured, keeping in mind that that one of our servers would provide web traffic. These security configurations would ensure that our traffic turned from using HHTTP to HTTPS. To avoid paying more fees for third party

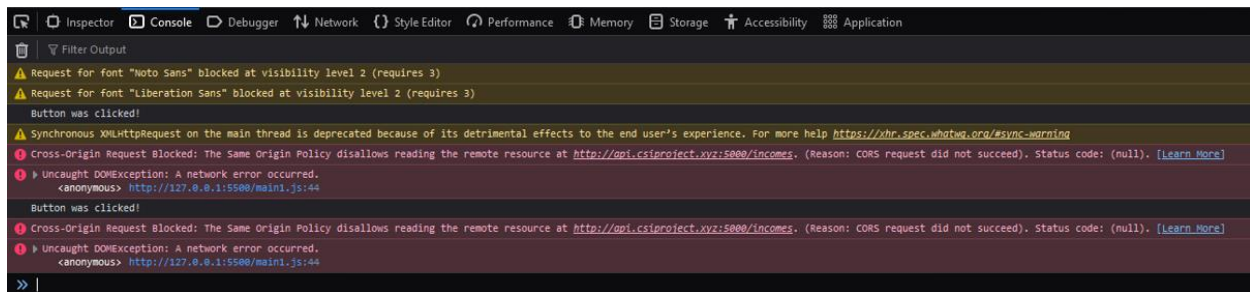
certificates, self-signed certificates were explored. A self-signed certificate was set up on the webserver but, during development testing, the browser provided warnings regarding self-signed certificates. This would not be acceptable to create a user-friendly interface. Also, basic use of encryption for secure communication felt essential given the project was related to the cyber security field. Further search led our team to explore certificates assigned by Let's Encrypt, which is a third-party Certificate Authority. A program named Certbot would allow our team to have certificates assigned by this CA.

The Certbot guide was used to set up Let's Encrypt Certificates and enable HTTPS traffic on port 443. We used our Linux OS to set this up, targeting our webserver application, Apache.



Later, when linking the website to our API using front-end technologies like JavaScript, our team would encounter errors retrieving data. The errors were related to a concept named CORS (Cross-Origin Resource Sharing). These errors were explored in more detail using Developer Tools for the web browsers. To remove this issue the API server was configured similarly to the Web server. That is, the same domain name was assigned to it through a subdomain creation. Certbot was also configured on the API server such that both servers communicated with the client using HTTPS on port 443. The API server used Flask configuration instead of Apache to reference the third-party certificate.

Some errors which were removed during development are shown below.

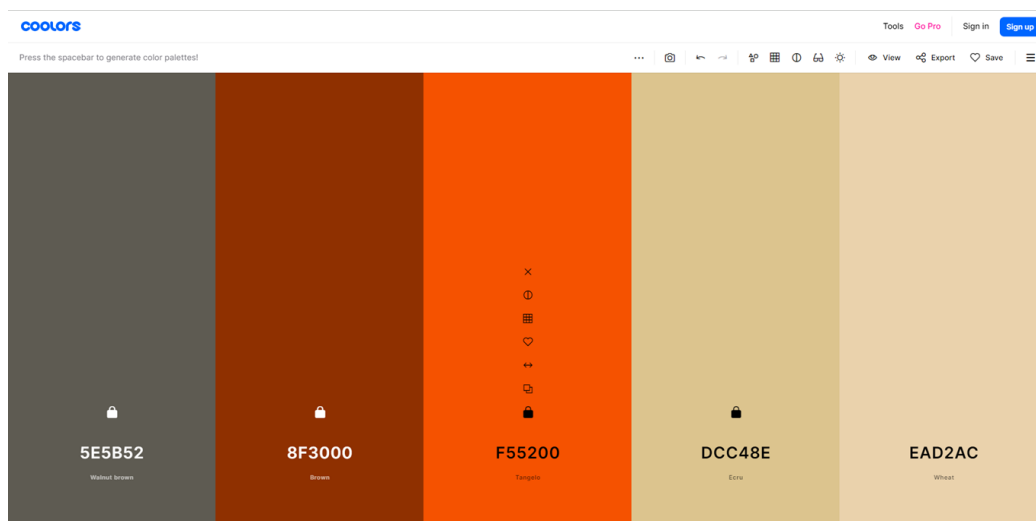


The main results from this subsection were implementing secure communication using a Certificate assigned by a third-party Certificate Authority.

Webpage Design

The original design of the webpage was improvised while learning front-end web technologies – HTML, CSS, and JS. Some key design concepts were kept in mind during the implementation: Simplicity and Consistency. The design process had to go beyond code editors to get an overall idea of how this graphical interface would look. Some initial steps of this process involved looking at color schemes, to support visual harmony.

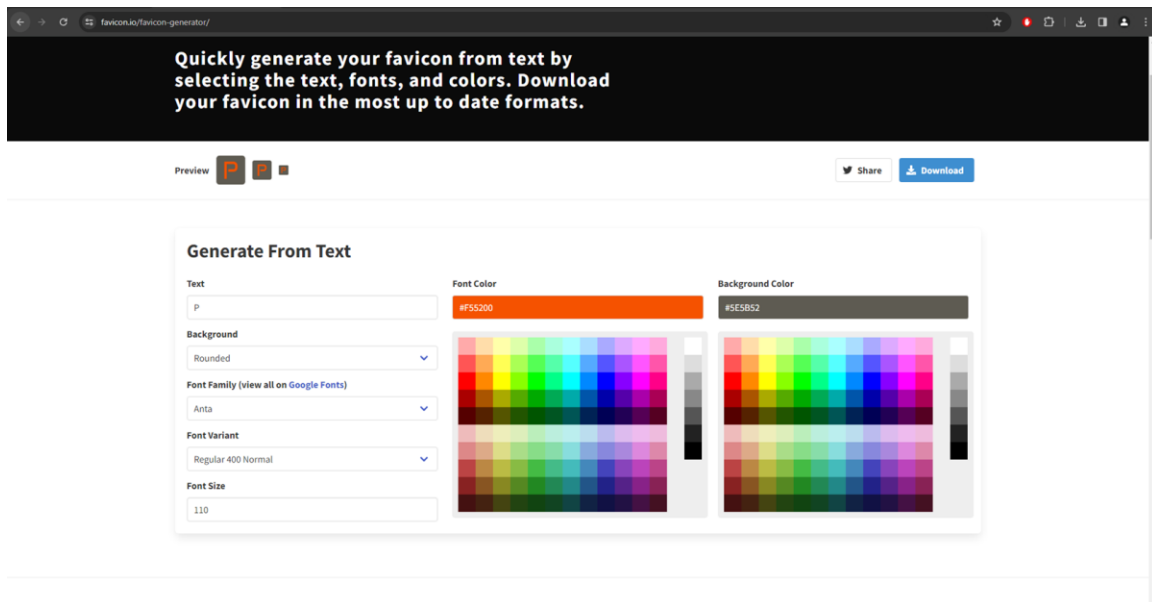
Below is a view of the original color palette picked for the first design of the webpage. Orange was to be the accent color and neutral colors would be supporting colors. These choices later changed through the development process.



An interesting consideration identified during the web development process was that everything which was visible had to be defined somewhere. Even simple websites can require a lot of code for the variety of elements that they present to the client. Tiny details

like the favicon image would be visible to users. There was a high amount of attention to detail required for this process and a lot of edits were required.

Tools like a favicon website were used to generate details that provided a more consistent look to our webpage.



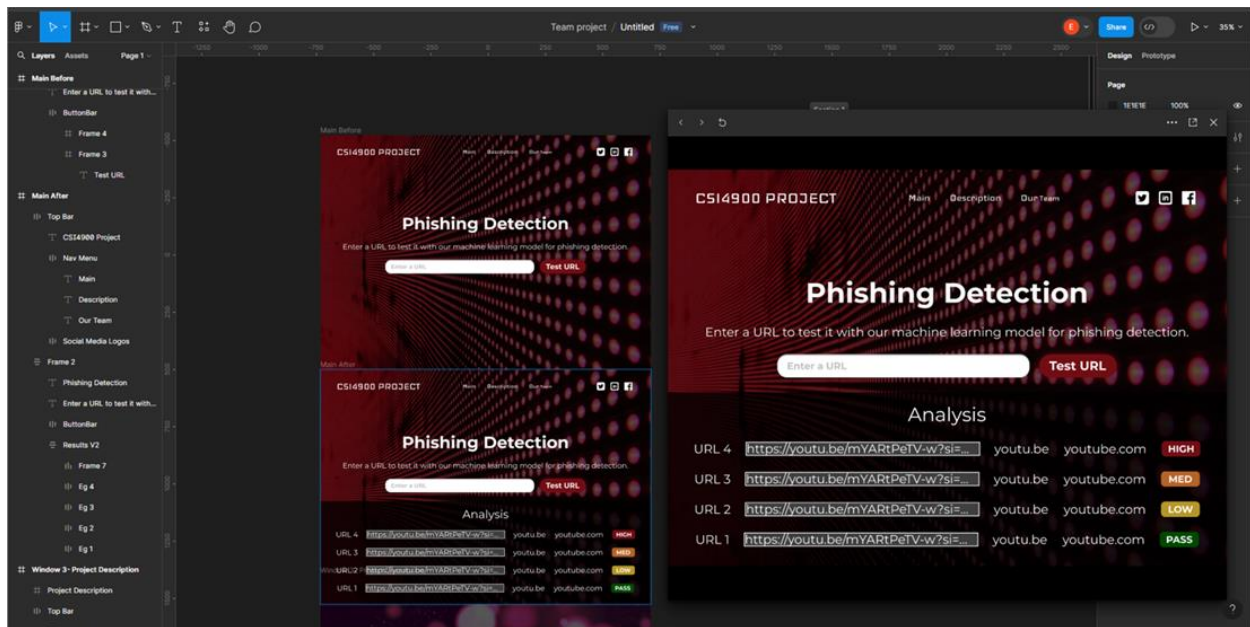
Below is a view of an early version of the webpage before exploring prototyping and frameworks.



The process of building a website without a clear direction is a time-consuming process that includes many revisions. The timing requirements for this project required a different approach. Our team became familiar with design and prototyping tools that allowed us to

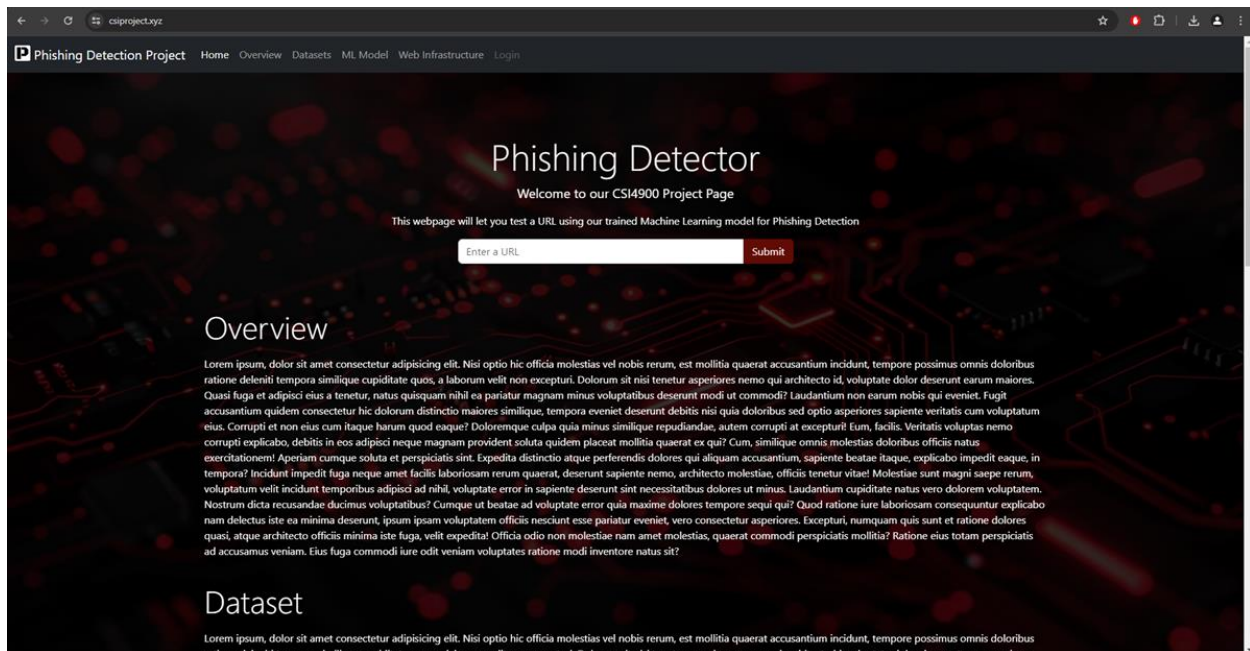
focus on the visual aspects and the interactivity of the webpage rather than the coding. The main tool used to create the main design is called Figma.

Below is a view of the prototype webpage as seen during the design process. The prototype had a more inviting look compared to earlier designs and the interactivity offered by Figma provided a clearer picture of how to build the webpage.

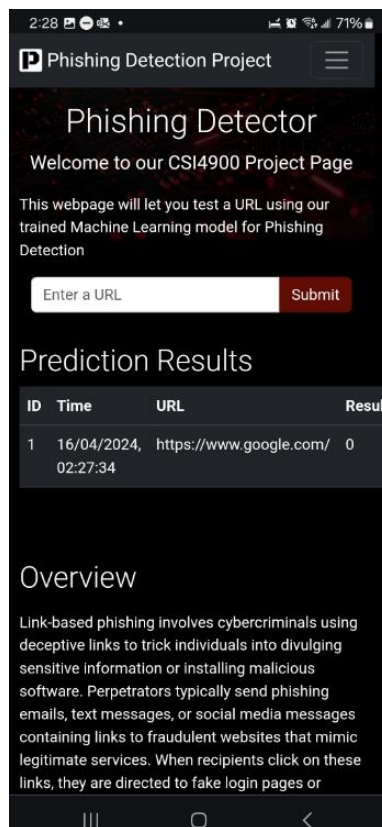


As seen on the left pane in the Figma interface, there are many details to implement. During the following process of implementing the design through code, the code started to become complex as several elements required their own manual formatting using CSS. To aid our team with this challenge, the Bootstrap framework was explored which facilitated the styling of elements by providing classes which were configured for style and responsiveness.

After using the framework, our webpage held a much cleaner and presentable look which was maintained across different screen sizes through responsiveness.



Above is a view of the webpage after introducing the Bootstrap framework as seen through a computer's web browser. Below is the view as seen through a phone screen.



Sections of the webpage were created to give some brief information to users about what was done for our project.

The main section of the webpage involved an input bar which allowed a user to provide a URL that the web client would then send to the API server to provide a prediction. To provide more value with the space that the webpage offered, the webpage allowed multiple past results to be visible by presenting them in a table format. This would make the interface more user friendly by removing the need for users to remember the prediction outputs if they were comparing links. The prediction history was however not stored anywhere besides locally through the user's current active session.

API Setup

The technologies selected to provide the API functionality were chosen with respect to the other work performed by the rest of the team. Since the model was created using Python and the scikit and joblib libraries, the software chosen to provide the API was Flask.

Flask is a Python framework that allows the creation of APIs. Each endpoint, which can be referenced in a web browser URL bar, can be programmed to provide specific functionality.

A significant endpoint that needed to be configured involved a function which given a URL as string, provided an array of features as required by our model to make predictions. We named this process the feature extraction process. Another significant endpoint was created to provide users with a prediction, given a URL. This included the feature extraction process and then using the resulting array with the loaded model to make predictions. These API calls were of the type POST as they provided our server with data and required a response.

The model was loaded into memory at the start of the Flask script so that it only had to be loaded once, rather than every time a prediction was necessary.

The required packages were installed with the pip app using the bash command line. Some packages that were used were *scikit-learn*, *dnspython*, *pandas*, *tranco*, *request*, *socket*, *beautifulsoup5*, *whois* and *tld*. Many of these were mostly used for the feature extraction process.

The pyflakes package was used to check the python code for errors during the development process which mainly occurred using the Linux terminal through the vim text editor.

Below is a screenshot of the details provided by Flask when it is started showing that it is listening for https traffic on port 443.

```
[root@csiprojapi pythonproj]# python api.py
* Serving Flask app 'api'
* Debug mode: on
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on https://127.0.0.1:443
* Running on https://155.138.142.49:443
INFO:werkzeug:Press CTRL+C to quit
INFO:werkzeug: * Restarting with stat
WARNING:werkzeug: * Debugger is active!
INFO:werkzeug: * Debugger PIN: 515-090-645
```

Some calls to the API endpoints were able to be done exclusively using the URL bar of the web browser. Below is an example of a test used to extract the features of a malicious URL.

[illegible]

The prediction function returned a string with a value that represented whether a URL was predicted as malicious or not. Malicious URLs would return a string with a 1, while benign URLs returned a string with a 0.

← → ↻ 🌐 api.csiproject.xyz/predicturl?urltotest=https://managehosting-rinnovare-domini.servital-ma.com/aruba/AreaUtent

[1]

The actual endpoints were then referenced by the web client as provided by the JavaScript code from our Web Server.

Successes

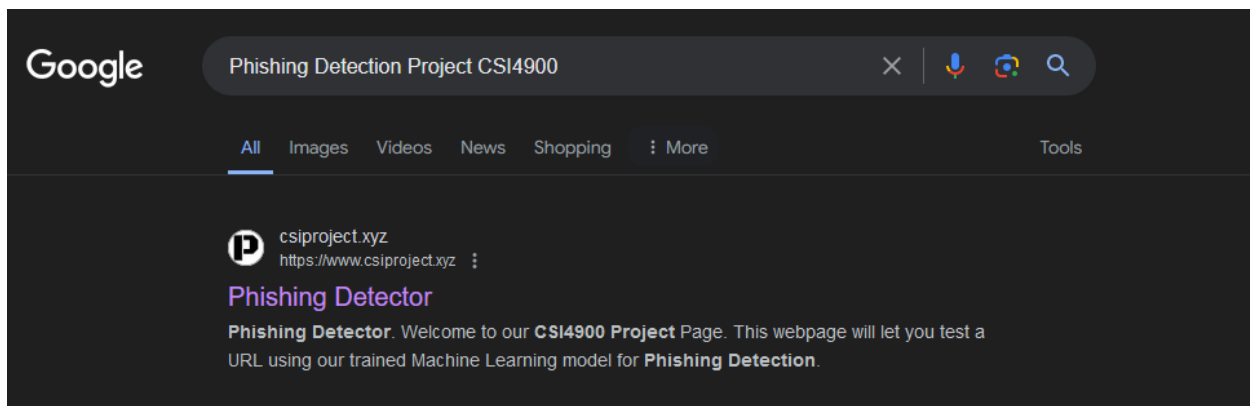
The experience of performing the Web Infrastructure section was rewarding. It provided an accessible and clean interface for users to interact with our project. Our team was able to learn about web development. The result is functional and uses encryption to provide secure communication with our servers.

The screenshot displays the 'Phishing Detector' web application in a browser. The main interface includes a header with navigation links (Home, Overview, Datasets, ML Model, Web Infrastructure, Login) and a central prediction form with a 'Submit' button. Below the form, a 'Prediction Results' table shows two entries:

ID	Time	URL	Result
2	4/15/2024 1:54:35 AM	https://en.wikipedia.org/wiki/Linux	0
1	4/15/2024 1:54:29 AM	https://csiproject.xyz/	0

Two red arrows highlight the 'Time' and 'URL' columns of the results table. The bottom portion of the image shows the browser's developer tools, specifically the 'Network' tab, which lists several requests to the application's endpoints. A red arrow points to the 'predicturl' request, which is a GET method to 'api.csiproject.xyz/predicturl' with a status of 200. The console also shows some warnings related to font loading and deprecated API usage.

Our team learned about the various technologies that are used to provide a web service and the accessible cost to get started. A pleasant surprise was to see our website identified by the Google search engine as shown below.



Section Summary

For the web infrastructure of this project two servers were created. The web server used Apache and front-end web technologies such as HTML, CSS, and JavaScript. The API server provided back-end functionality through Python functions presented as endpoints by the Flask framework. Both servers were hosted on a Toronto datacenter and used third party certificates to provide secure communication.



Results and Discussion

The implementation of a phishing detection model utilizing machine learning algorithms, such as the Random Forest classifier, yielded promising results with an accuracy of 96% and a Matthews Correlation Coefficient of 91%. Feature analysis techniques such as VIF, LIME, SHARP, and Permutation Importance provide valuable insight into the importance of different attributes originating from the team's feature engineering or from the reviewed literature. Further improvement of the dataset is possible through the enrichment of features related to HTML and JavaScript content of the pages the URL led to, as opposed to just the features extracted from the string URL and the querying of more information utilizing the parsed information.

Challenges encountered during the training process, including overfitting, underfitting, and limitations of accessing external APIs, were addressed through careful optimization strategies, along with the refining of the dataset, and ensuring comprehensive feature selection. The model achieved a high level of accuracy while mitigating the risks. Finally, the final implementation, packaged into a .joblib file, allows for seamless integration into web servers, and represents an advancement in phishing detection tools that are reliable and efficient accessible by users to safeguard against online security threats.

Next Steps

Our team's next steps involve creating the web browser extension so that our model can provide predictions while a user is browsing their e-mails. This would allow the users to receive warnings about URLs, which could be malicious to their system or cause other types of personal harm, without the need to open another tab and carefully copy the URLs. To do this our team is required to learn web development in terms of developing a browser extension. Further exploration is required to understand privacy concerns as an extension that would be able to read the link content of an e-mail would be able to read the other content present in the e-mails.

Our current dataset is limited by the string information we can extract from each URL, and all the information we can query from that. Further improvement could include scraping the HTML and JavaScript details from each URL to provide more insight that could be valuable in determining between malicious and safe URLs. Extracting HTML tags from the webpage could prove useful as metadata and script tags can be useful in determining malicious activity, as well as the presence of code obfuscation within JavaScript code, which aims to make the code unreadable to users but can still compile and run regularly.

Our team is also considering participating with the project's supervisor in authoring a research paper regarding our work and our findings. Further communication should be arranged to understand the requirements of this task.

Conclusion

By designing and implementing a tool using an AI model with machine learning techniques for phishing detection, we aim to effectively combat the ever-evolving cyber threats that impact individuals and businesses. This tool not only enhances user security but also offers valuable insights to help navigate and address the challenges of today's digital landscape.

References

- Federal Bureau of Investigation. *Internet Crime Report*. Internet Crime Complaint Center, 2021, www.ic3.gov/Media/PDF/AnnualReport/2021_IC3Report.pdf.
- Mahajan, Rishikesh, and Irfan Siddavatam. "Phishing Website Detection using Machine Learning Algorithms." *Research Gate*, vol. 181, no. 23, Oct. 2018, <https://doi.org/10.5120/ijca2018918026>.
- Gana, Noah N., and Shafi'I M. Abdulhamid. "Machine Learning Classification Algorithms for Phishing Detection: A Comparative Appraisal and Analysis." *Research Gate*, Jan. 2020, <https://doi.org/10.1109/NigeriaComputConf45974.2019.8949632>.
- Upadhyay, Akriti. *Medium*, Medium.com, 23 Oct. 2023, medium.com/@akriti.upadhyay/phishing-detection-met-generative-ai-365b3e89920d.
- Upadhyay, Akriti. *Kaggle*, Kaggle.com, 23 Oct. 2023, www.kaggle.com/code/akritiupadhyayks/phishing-detection-with-machine-learning?scriptVersionId=147660313.
- Kakarla, Swaathi. *Activestate*, Activestate, 11 Feb. 2021, www.activestate.com/blog/phishing-url-detection-with-python-and-ml/#:~:text=Since%20the%20dataset%20contains%20boolean,models%20work%20best%20for%20classification.
- *Scikit-learn*, Scikit-learn, scikit-learn.org/stable/data_transforms.html.
- *Kaggle*, Kaggle, Dec. 2023, www.kaggle.com/code/dima806/phishing-email-detection-distilbert-huggingface/notebook.
- Xu, Pingfan. "A Transformer-based Model to Detect Phishing URLs." *Arxiv*, 5 Sept. 2021, www.kaggle.com/code/dima806/phishing-email-detection-distilbert-huggingface/notebook.
- Worch, Mordechai. *Ironscales*, Ironscales, 23 Sept. 2020, ironscales.com/ironscales-engineering-corner-blog/developing-a-machine-learning-model-to-identify-phishing-emails/.
- Büber, Ebubekir. *Towardsdatascience*, Towards Data Science, 8 Feb. 2018, towardsdatascience.com/phishing-domain-detection-with-ml-5be9c99293e5.
- *Phishingbox*, Phishingbox, www.phishingbox.com/resources/articles/evolution-of-phishing-attacks.
- Zvornicanin, Enes. *Baeldung*, edited by Grzegorz Piwowarek, Baeldung, 18 Mar. 2024, www.baeldung.com/cs/ml-feature-

[importance#:~:text=All%20we%20need%20is%20to,correlation%20coefficient%20are%20more%20important.](#)

- Trevisan, Vinícius. *Towardsdatascience*, Towards Data Science, 17 Jan. 2022, towardsdatascience.com/using-shap-values-to-explain-how-your-machine-learning-model-works-732b3f40e137.
- Yasin, Adwan, and Abdelmunem Abuhasan. "AN INTELLIGENT CLASSIFICATION MODEL FOR PHISHING EMAIL DETECTION." *International Journal of Network Security & Its Applications (IJNSA)*, vol. 8, no. 4, July 2016, <https://doi.org/10.5121/ijnsa.2016.8405>.
- Vrbancić, Grega, et al. "Datasets for phishing websites detection." *Data in Brief*, vol. 33, Dec. 2020, <https://doi.org/10.1016/j.dib.2020.106438>.
- Sriva, Tavish. *Analytics Vidhya*, Analytics Vidhya, 22 Aug. 2023, www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/.
- Chu, Lan. *Towards AI*, Towards AI, 21 July 2022, pub.towardsai.net/model-explainability-shap-vs-lime-vs-permutation-feature-importance-98484efba066.
- Hulstaert, Lars. *Towardsdatascience*, Towards Data Science, 11 July 2018, towardsdatascience.com/understanding-model-predictions-with-lime-a582fdff3a3b.
- Selvaraj, Natassha. *KDnuggets*, KDnugget, 5 Oct. 2022, www.kdnuggets.com/2022/10/hyperparameter-tuning-grid-search-random-search-python.html.
- Chandra, Yuvraj. *Makeuseof*, Makeuseof, 14 June 2023, www.makeuseof.com/regular-expressions-validate-url/.
- *StackExchange*, StackExchange, Sept. 2019, stats.stackexchange.com/questions/365778/what-should-i-do-when-my-neural-network-doesnt-generalize-well.
- Lawton, George. *TechTarget*, TechTarget, 9 Aug. 2023, www.techtarget.com/searchenterpriseai/tip/Types-of-learning-in-machine-learning-explained.
- VULTR. *Introduction to VULTR DNS*. Vultr Docs, Vultr, 13 Jan. 2022, docs.vultr.com/introduction-to-vultr-dns.
- Digital Ocean. *Point to Digitalocean Name Servers from Common Domain Registrars*. DigitalOcean Documentation, docs.digitalocean.com/products/networking/dns/getting-started/dns-registrars/. Accessed 24 Apr. 2024.
- Certbot Project. *Certbot Setup for Apache on CentOS*, certbot.eff.org/instructions?ws=apache&os=centosrhel7. Accessed 24 Apr. 2024.
- Hannousse, Abdelhakim. *Web Page Phishing Detection*. 3rd ed., Mendeley Data, 2021.
- Vrbančič, Grega. *Data in Brief*. Vol. 33, Science Direct, 2020