Improving the Efficiency of a New Malicious Domain Prediction System

by

Aashish Arora

B.Eng, Gujarat Technological University, 2019

A Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

University of Victoria

Supervisory Committee

Malicious Domain Prediction System

by

Aashish Arora

B.Eng., Gujarat Technological University, 2019

Supervisory Committee

Dr. Fayez Gebali, Department of Electrical and Computer Engineering
Supervisor

Dr. Issa Traore, Department of Electrical and Computer Engineering
Supervisor

Abstract

Cybersecurity is a key concern in today's digital era and healthy number of cyber-attacks are launched every day. Malicious domains represent one of the media through which attacks are launched and malicious artifacts are spread. While many malicious domains are known and blacklisted, a sizable number of new domains registered by cybercriminals are unknown to blacklist maintainers, and as such can be used undetected in ongoing and future hacking campaigns. The Domain Prediction System (DPS) is a prototype malicious domain prediction system developed by one of the industry partners of the ISOT Lab. Based on a small number of seed blacklisted domains, DPS generates a list of associated registered domains that can potentially be malicious in the future. Predicting malicious domains is a long slog process that involves mining and iterating over billions registered domains. This project focuses on reviewing, evaluating, and improving the performance of the prototype implementation of DPS. A code was provided but had several efficiency issues and inaccurate outputs. As a result, this report identifies problems in the existing code and proposes solutions to improve performance. Additionally, some experimental details are presented to demonstrate effectiveness. Furthermore, a Flask web-based application was developed to host the project and make it easier to use.

Table of Contents

List of Tables

List of Figures

Glossary

- Cybercriminal: A person that is involved in performing criminal activities using computers and internet.
- DNS: Domain Name System
- ICANN: Internet Corporation for Assigned Names and Numbers
- Malware: A computer program that is developed to interfere, damage or gain unauthorized access to a computer or network.
- Original Code:
- TLD: Top-Level Domain

Acknowledgements

I would like to express my gratitude and appreciate Dr. Issa Traore for providing me with valuable guidance and support throughout the project. Their expertise, insights and encouragement helped me to develop ideas and refine the project.

I would also like to thank Dr. Fayez Gebali for inspiring me to believe in my capabilities and bless me with numerous opportunities throughout my degree.

Chapter 1: Introduction

1.1 Cybersecurity and Malicious Domains

Information security in today's digital world is extremely important given the opportunities and resources available for cyber criminals. Cybersecurity is an area of information security that involves the protection of networks, servers, computers, and data against unlawful accesses and activities. The three major concerns of cybersecurity include confidentiality, integrity, and availability.

Malicious domains are used to host websites that are built with the intention of carrying out fraudulent activities, such as phishing schemes, pirated software distribution, and data theft. These fraudulent attacks can be targeted at individuals or big corporations, with different levels of severity and logistics. Malicious domains provide an anonymous and easy way to carry out Internet-based attacks. This is because malicious websites are created with the intention of duping the victim into believing that the website, they are visiting is legitimate. Thus, by creating malicious websites that resemble a legitimate website, Internet users fall victim to cybercriminal activities. Cybercriminals use malicious domains to lure victims into clicking further malicious links or trick users into entering their credentials, which can be used to gain unauthorized access to their personal accounts.

On the other hand, malicious domains are also widely used to distribute malware. Devices infected with malware may get affected by disruptions of regular services and can also transmit data from the victim's device to the attacker's device, resulting in unauthorized access to the data. Distributed denial-of-service (DDoS) attacks, which flood a website with spurious data to make it inaccessible to genuine users, can also be carried out using malware.

1.2 Problem Statement

Continuous development in today's technological world results in increasing risks of cyber-attacks. Malicious domains, in this case, help to spread malware, steal sensitive data, and provide phishing capabilities to cybercriminals. Threat actors register malicious domains with names that are highly similar to the legitimate domains, thus making it very hard for individuals as well as organizations to identify them. Also, it is very hard to mitigate the registration process for these kinds of domains since the content on the website might seem harmless, but can have harmful links or intentions. For that reason, it is important to predict and detect malicious domains before they can be used for any bad practices. After prediction, the domains can be monitored for future changes to detect any malicious activity and report it for a takedown or blacklist. A prototype domain prediction called Domain Prediction System (DPS) was developed at the Information Security and Object Technology (ISOT) Lab. By inputting an existing blacklisted domain, DPS generates a list of (predicted) domains that have the potential to spread malicious content similar to the blacklisted domain. The original version of the DPS has several known performance issues.

The objective of this project was to test and improve the efficiency of DPS based on available Python code snippets, further referred to as original code in this following report. The report focuses on improving the performance of the available original code and thus enhancing the speed and output of the code

1.3 Report Outline

This remaining chapters in the report are structured as follows.

In chapter 2 the background, dataset and problems in the original code are discussed. Further in chapter 3, suggested improvements to the different parts of the code are presented. Next, in

chapter 4 the improvement results are shown. Finally, we conclude the report in chapter 5 with some future work recommendations.

Chapter 2: Background

2.1 Dataset and Data Source

DPS uses as input an existing blacklisted or known malicious domains and produces a list of potentially malicious domains related to the input domain. To do that it relies on the database of registered domains. Such database is maintained by the Internet Corporation for Assigned Names and Numbers (ICANN).

ICANN coordinates unique identifiers for the Internet, like a person or number. ICANN helps to provide critical services for the Internet's need to work based on the address book called the Domain Name System (DNS). ICANN defines policies for 'names and numbers' and how they should work [1].

For this project, we have used a text file called com.zone that contains a list of all the domain names that have been registered under the .com top-level domain (TLD). These root files are managed by ICANN, which contains information about all TLDs like .com, .in, .au, .ca and technically all existing TLDs [2]. DNS software uses this file to enable domain name resolution for websites and all other online resources situated under the .com TLD. The columns available in the com.zone file are URL (domain), time to live (TTL), domain name system security extensions, name server type, and name server host.

The com.zone file used here has certain properties, like the fact that the dataset is alphabetically sorted by the first column, which is the domain name, and there will be multiple entries for the same domain if they are hosted by different nameservers. Figure 2.1 shows a very small snippet of the com.zone file.

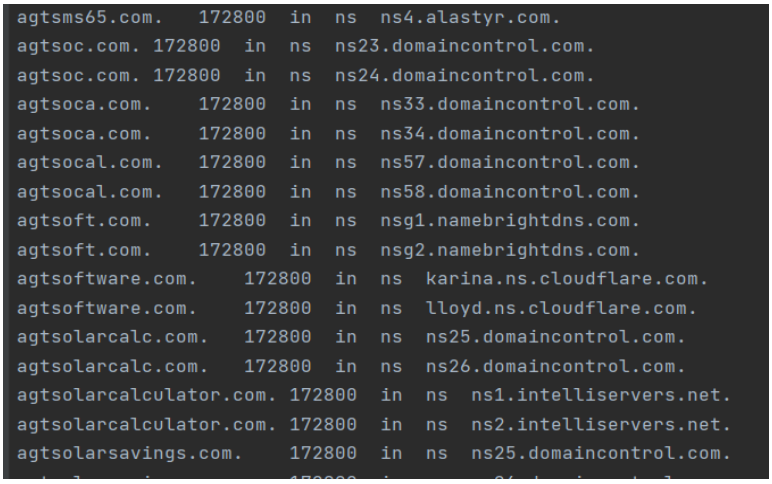

Figure 2.1 Snippet of com.zone file

2.2 Working of the original code

Figure 2.2. outlines the working and building blocks of the original DPS (i.e., before the project). An existing code for DPS was provided based on server and client architecture written in the Python language. This code involves processing from the server end of the program and use the client end for user interaction. The code was divided into three major sections: iterating over the dataset (com.zone) file, filtering domains using existing blacklist and whitelist databases, and finally executing the code.

Starting with the 'predict' function, it reads data from the zone file and performs some operations to start finding the submitted (i.e., input blacklist) domain and potentially list down related domains. This function returns a JSON dictionary with the seed domain (the submitted domain) as well as a list of domains that might be malicious in the future. If the seed domain is not present in the zone file, it will stop the code from here and return a string indicating the same.

In the second part of the code, after taking the list of domains from the first part, the function will try to find more information using WhoIs API call using a Python library. A built-in Python function called gethostbyname() will translate a hostname to its corresponding IP address and

is used to prove the life of the domain and its existence. The third part of the code filters out the list of domains from existing domains blacklists and whitelists, which are updated regularly. Lastly, a list of potential domains is generated that can be converted to malicious domains in the future but are not currently blacklisted.

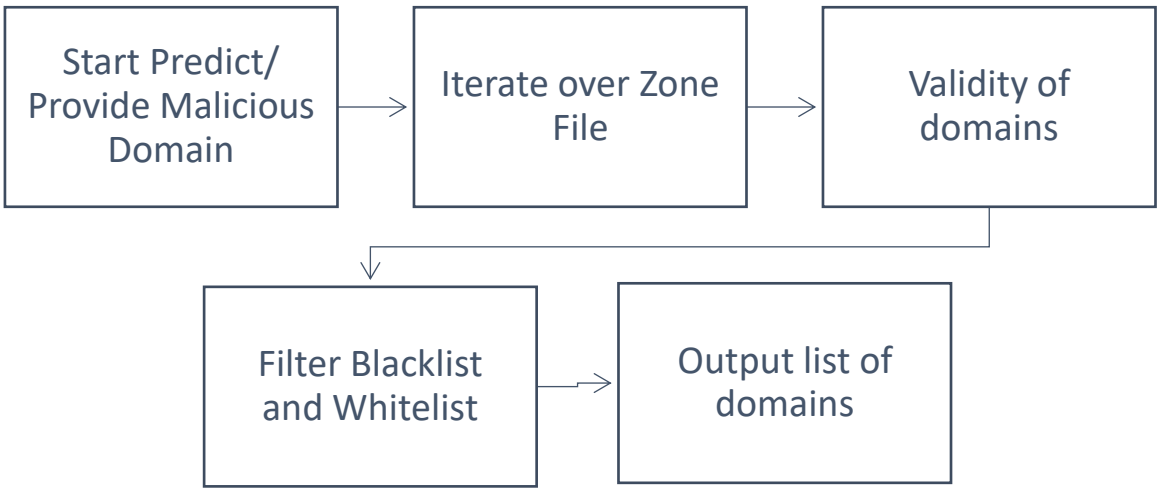

Figure 2.2 Flowchart of original code

2.3 Issues and errors in the original code

The provided code had a few issues including no outputs on certain inputs, including the following:

a. Inefficient processing: The processing of the code is very slow since it needs to read through a large zone file (approximately 2.3 Gb) each time it is run. Since this is an expensive operation, it causes delays in processing and is different for each unique input. On top of it all, there are multiple iterations for each input on the large zone file.

b. Single process: The code is designed to run on only one process; thus, this excludes the benefit of utilizing multi-core processors on the computer. This also slows down the process of searching and sorting a large zone file.

c. Inconsistent output: The code would produce contrasting outputs for the same input on multiple trials, which makes the output unreliable. This could be due to bugs in the code or network conditions while checking the existing blacklists and whitelists.

d. Slow filtering APIs: The code's filtering on BL, WL, DNS blacklists, and DNS whitelists is extremely slow due to reasons like low network bandwidth, a single API call, and a large amount of data input/output.

In the following chapter, proposed solutions for the aforementioned problems are discussed, and their implementations are discussed along with achieved performance improvements.

Chapter 3: Proposed Improvements

3.1 Database management and chunking

In the original code, the zone file was implemented as a text file that was used iterating over it line by line. To resolve this issue and improve processing efficiency, we worked on making small chunks of the large file and processing only what is needed to reduce the time of processing. In the original code, there are two major iterations on the zone file: the first is to find the submitted domain, and the second is to find all other domains hosted by the same nameservers as the submitted domain.

For the first iteration, since the zone file is alphabetically sorted, we made chunks of rows starting with the same alphabet and saved them in different files. Furthermore, we saved all those files into one folder named ‘chunks_of_urls’. Thus, searching the submitted domain became easier and faster by loading only a small file consisting of the domains starting with the same first letter as the submitted domain.

For the second iteration on the dataset, we had to find all the domains hosted by the same nameservers as the submitted domain. Since the nameservers were not sorted in any manner, a secondary code was developed to address this issue, which is explained in the following section. Small chunks of the zone file were created to facilitate faster processing and searching for relevant domains. The files were created based on the first 3 letters of the name server and saved in a folder named "chunks_of_ns". Thus, searching the nameservers found hosting the submitted domain became faster by only loading the relevant nameserver file and saving it to a list for further processing.

The standalone "chunking.py" code was written in Python and has the goal to split zone files into smaller chunks, write them in different files, and save them in relevant folders. The code initializes by defining two directories named "chunks_of_urls" and "chunks_of_ns" for two

different sets of files as discussed above. Further, the first chunking operation reads the zone file line by line and splits its components to check if name servers for each domain are present. If available, it extracts the first character of the first component (i.e., the domain name) to use as a filename and then appends the current line to the same file. To make the write operation faster and more efficient, we used a dictionary and counting scheme where, after a certain number of rows are iterated, a single write command will append each domain to a file with a corresponding filename in the dictionary and then reset the counter to zero. The code repeats this process for all lines in the zone file.

The second iteration in "chunking.py" is like the first iteration discussed above but uses name servers as the primary key. Here, the first letter of the nameserver is considered and appended to the corresponding file where the filename starts with the same letter. This part also follows the same dictionary and counting method to make the process faster. Here, after a couple of trials and errors, we noticed that the file size for nameservers starting with "n" was large and occupied just over 75% of all the files. Thus, to reduce processing on this file, the same concept of chunking was applied to filename "n". Further, the best option to process the system faster was to divide these name server lists by using the first three letters in the account and saving them to corresponding files.

Figure 3.1 shows how the old system processes the dataset iteration in the code. Here, as mentioned in the previous chapter, the code will iterate sequentially over the com.zone file for 2 times. Figure 3.2 shows the flowchart of how the chunking.py file works and how it is used in the improved system. The first part of the figure shows how the chunks are formed and the second part of the figure shows at what stages the improved code accesses the chunks.

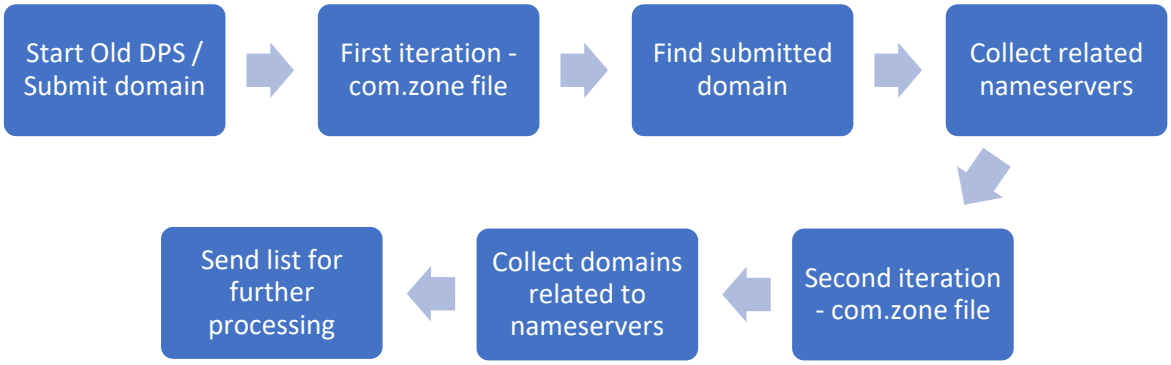

Figure.3.1 DPS (Old Code) First Part - Iteration over com.zone file

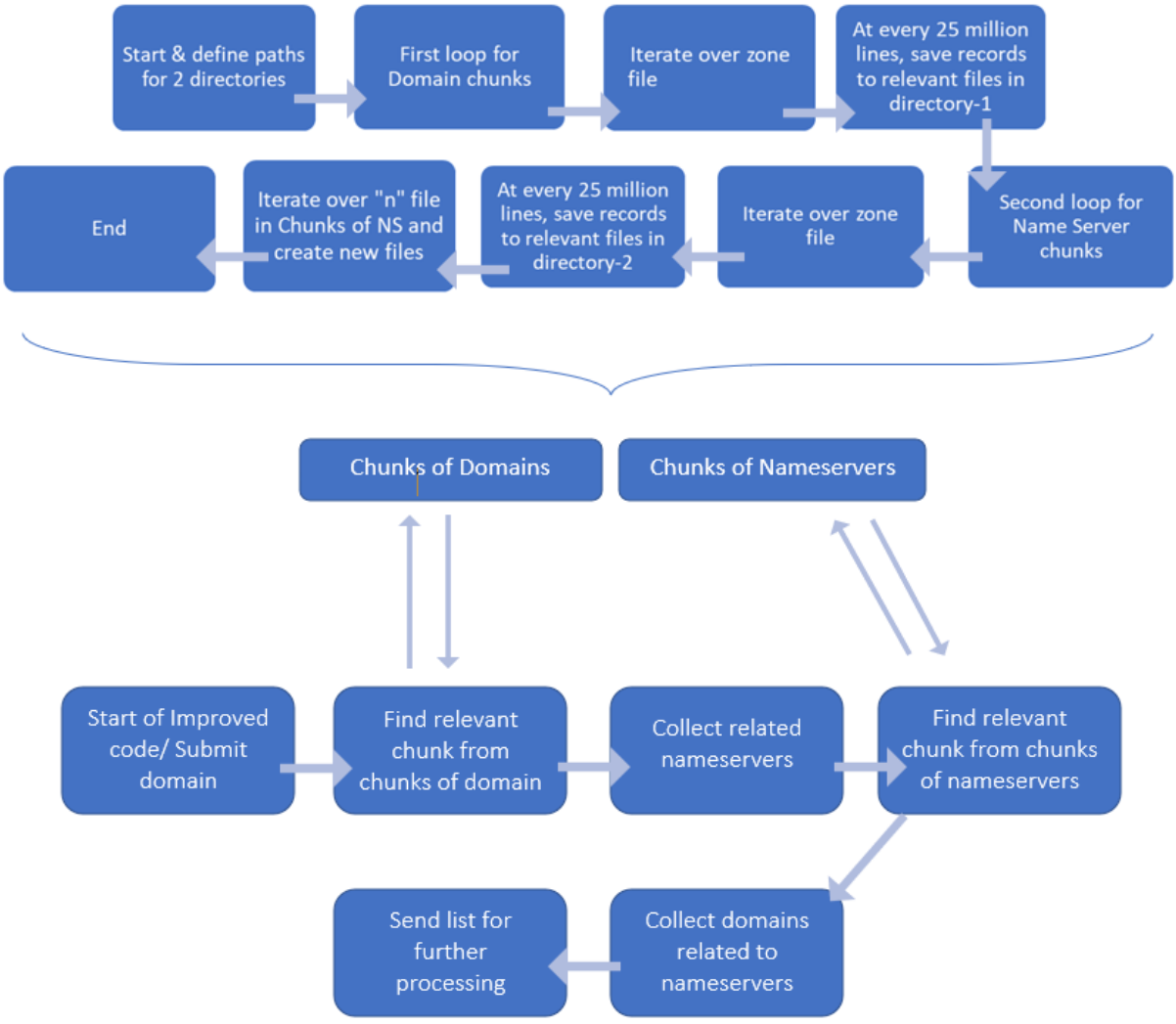

Figure 3.2 Blocks of chunking.py and how chunks are used in improved code

3.2 Adding Multiprocessing in the WhoIs module

Another issue with the original code was slower processing due to the single process used to perform all the operations. One of them was to check for “whois” information for all the domains passed from the iterations over the data. Here, we used Python’s “concurrent.futures” module to run tasks in parallel. Considering this scenario, the list of domains would be iterated by each available executor (i.e., each process) and processed as per the function call mentioned. The function mentioned here is "socket.gethostbyname()”, which will check the validity and liveliness of the domain.

Python’s “concurrent.futures” module provides a high-level interface for asynchronously executing function calls as per the need. “ThreadPoolExecuter” or “ProcessPoolExecuter” are classes used, depending on the context, to use multiple threads or processes, respectively. In our case, we used the “Executor” abstract class and its objects. “Executor” is a common interface for both “ThreadPoolExecutor” and “ProcessPoolExecutor” with a couple of main methods like “map()” and “submit()”. The function “map()” was used, which asynchronously executes the function on the iteratable list of domains and returns a “Future” object [3].

This part of the code starts with a function definition, “whois()” which takes a list of domains as input from the previous part of the code, and converts it to a JSON object to extract domain names. The extracted domain names are passed to the next function called “get_whois()” to perform a WHOIS lookup. The get whois() method uses the socket library to resolve the IP address of the domain name in order to determine whether it is genuine. If the domain name is valid, the code returns the domain name; otherwise, it returns “None”. Using a pool of processes, the get whois() function is executed concurrently by the whois() method using the "concurrent.futures" module. When every procedure is finished, a JSON string called “whois json” is created and returned for further processing [4].

3.3 Clustering on filtering API calls

In the code, the forwarded list of potential domains is to be filtered for any existing blacklisted or whitelisted domains. There are in total 4 different lists used to do this process: a blacklist, a whitelist, a DNS blacklist, and a DNS whitelist. The domain blacklist is a list of domains that are restricted from being accessed or blocked by browsers. This is to restrict users from accessing spam, malware, phishing, and other websites that host malicious content. Similarly, DNSBL is a list of IP addresses and domains that are included in popular blacklists like Spamhaus Block List (SBL) [5]. This list is created and updated based on the regular scanning of newly registered domains and existing domains for any malicious activities by different companies like Google and Spamhaus.

On the other hand, whitelists and DNSWL are the lists used by companies to make a trusted list for their specific use. These are used to avoid blocking of domains as well as emails from specific sources. Also, whitelists are used to restrict access to specific websites or services. Similarly, DNSWL is used to allow users to bypass certain security restrictions. In conclusion, DNS blacklists and whitelists are used exclusively for DNS servers, while blacklists and whitelists are used to limit or allow access to particular domains or IP addresses.

In our case, these lists are accessible through APIs made available by an industry partner of ISOT [6]. A method called “requests.post()” is used to make an HTTP POST request to a specific URL. This API requires a JSON file, a dictionary in our case, to query the list according to a specific method like blacklisting or whitelisting. This dictionary contains the requested data to be sent as a payload to the API. The file has to be in the specific dictionary format, like: data = {“urls”: batch}, where “data” is the dictionary that will be parsed for the API call and “batch” is the list of domains we need to submit. A response object is returned by the API request and can be stored in any variable. It would consist of the status code, headers, and the server’s response to the API call.

The code used in the submitted version can be explained as follows: A function defined as “filter_domain()” takes in two parameters: a method parameter specifying the type of filtering to perform (e.g., DNSBL or blacklist), and an input parameter containing a list of domains to be filtered. The function will return a filtered list of domains based on the method called. The function uses the “requests” library from Python to make API requests. In our case, to make faster and more efficient calls, a specific batch size of domains was decided and discussed with the API creator. The function separates the input into batches of 50 domains and sends API requests for each batch if the length of the domain list exceeds 50. The function gets the response from the API for each batch, removes any domains that it discovers, and adds the filtered domains to a results list. Lastly, the results list is flattened into a single list. This results list is passed to the next method call; that is, these filter calls will be made for each list one by one, starting with DNSWL, then whitelist, followed by blacklist, and finally DNSBL.

Chapter 4: Experimental Results and Improvements

4.1 Hardware setup used for testing

To perform the experiments and tests on the improved code a standard laptop was used. The experiments were conducted on a laptop with the following specifications: an Intel® Core ™ i7-7700HQ CPU with a base clock speed of 2.8 GHz and 8 cores, 16 GB of DDR4 RAM and an NVIDIA GeForce GTX 1060 graphics card with 6 GB of memory. The computer was running the latest version of Windows 10 Pro 64-bit as the operating system. The experiments were conducted using PyCharm Professional as the development environment and Python 3.9 as the programming language. The data was stored on the local HDD as per the need.

4.2 Improvements after the chunking of the zone file

To summarize, in the original code, the main concern was making the outputs efficient and fast to process. So, after developing the code with the proposed solutions stated above, there was a decrease in the processing time. There are a few items to discuss in this section.

a. Processing the Zone file using Chunking.py: Here, compared to the original code where the zone file was iterated over and over for every input, after chunking, saving the chunks is easier and more efficiently available to process. In terms of storage, there will be a need for double storage since the zone file is being chunked twice to produce chunks for domains and chunks for name servers, but the efficiency improvement is worth sacrificing the storage factor for. To be precise, the average size of the zone file is 24GB and would ideally require the same amount of disk space. But on the other hand, the average size of a chunk in a domain folder is only 510 MB, and the average size of a chunk in a name server folder is only 25 MB. But with the improved system, the ideal disk space would be double the size of the zone file.

Also, a further benefit of using chunking is the ease of updating the zone file; that is, whenever a new version of the zone file is available, it can be parsed into chunking code and produce the required chunks for processing.

b. Time taken to process: Initially, when the code would iterate over the zone file, it would take a fixed amount of time to run this segment. In this case, it was around 600 seconds on the test computer. On the other hand, when tested after new iterations and chunking, the last row in the largest chunk takes 100 seconds to compute, thus providing faster results.

   While computing, the time complexity for the original code would be $O(N^2)$, where N is the number of lines in the zone file. Here, the first loop iterates through each line of the zone file, and the same happens with the second loop. With respect to the studies, there were around 161 million ".com" TLDs reported in the second quarter of 2022 [7]. And the zone file available on ICANN that was used for this project contains around 406 million records, including same domains registered with different name servers. Thus, the time complexity will be huge for this process with such a large number of lines (i.e., N).

   On the other hand, after improvements in the code and chunking implementation, the time complexity became be $O(N \times M)$ where N is the number of lines in the domain chunk file and M is the number of lines in the corresponding name server files. Here, the largest file after chunking, in domain chunks is for the domains starting with "s". The size of the file is around 1.6 GB, and the total number of lines is 31 million, which makes this 92% faster than the original method. Similarly, for name server chunks, the largest file is for name servers starting with "ns1", with a size of 3.66 GB and 74.5 million records, which makes this part 81.5% faster than the original method.

Table 4.1 shows the experimental results based on the best, average and worst cases considering the improved way of data management (i.e., using chunking). Here the first domain is from the start of the first chunk of domains, the second one is from a middle file and the last domain is from the last part of the last chunk of domains file. The data in the table represents the time taken to run the dataset iteration by the code for both versions. So, the first domain should take the least amount of time in the first iteration of the data, followed by the second entry and lastly the third. It can be seen from Table 4.1 that the improved code resulted in a significant improvement in performance for all three cases. Compared to the old code, the improved code achieved an average improvement of 98.8%.

Table 4.1 Experimental results considering chunking.py implementation

Domain	Old Code (DPS)	Improved Code	% improvement
0-0-2-4.com	609.32 s	2.75 s	99.5 %
inkedmktg.com	605.29 s	6.36 s	98.9 %
zzzzzzzzzzzzzzzzzzz.com	611.06 s	12.95 s	97.8 %
Average	**608.55 s**	**7.35 s**	**98.8 %**

4.3 Improvements after multi-processing

For the improvements to the second part of the code, where WHOIS information is to be collected and verification of the liveliness of the domains is to be done, we introduced multi-processing. In the original code, there was only one process that was computing the list parsed from the previous section. It is a simple and straightforward way of iterating over a list. And would try to resolve the domain name to an IP address using the “socket” module of Python.

Besides, the multi-processing approach would provide faster processing on the list. Multiple cores from the available processor will call the “whois()” function on the mapped list. Each

worker process will try to resolve a domain name to an IP address and return the domain name if the resolution is successful. For example, while testing on a specific domain, it took just over 35 seconds to process a list of domains in the original code. On the other hand, after multi-processing implementation, it took only 3.16 seconds to process the same list on the same test computer. Similarly, on a new trial, a list took only 2 seconds to process with multi-processing but took around 9 seconds to process with only one process.

The processing time, in this part, also depends on the network traffic and bandwidth available to the computer. Also, the process will be faster with a dedicated processor and multiple processes running together. Thus, in theory, this will be as fast as possible as more processors become available.

Table 4.2 shows how the performance was improved after implementing multi-processing in this part of the code. By considering 8 blacklisted domains selected randomly as seeds (input), the following results were found. Here, it can be seen that the time taken is proportional to the number of domains generated from previous part of the code. Also, it can be seen that this part missed out on processing for certain domains while testing.

Table 4.2 Random experimental results after multi-processing implementation

Domain	**No. of Domains to process**	**Old Code (DPS)**	**Improved Code**	**% improvement**
camerquiz.com	553	14.11 s	2.1 s	85.1 %
judyguth.com	1553	83.8 s	9.5 s	88.7 %
blogspot.com	6081	717.7 s	89.3	87.6 %
oznemedya.com	7010	Couldn’t process	136 s	100 %
thecatbreeds.com	837	16 s	6.7 s	58 %
chitraprakashan.com	1007	49.1 s	7 s	85.8 %
drsridharspine.com	593	108.38 s	6.62 s	93.9 %
0m66lx69dx.com	6081	700 s	94.6 s	86.5 %
Average				**85.7 %**

4.4 Improvements after clustered API calls

In this part, the discussion will revolve around the filtering of the domain list with respect to ISOT industry partner's blacklist, whitelist, DNSBL, and DNSWL. Issues in the original code were related to inefficient processing and denial of service due to huge uploads to the API call. These issues were resolved by using the proposed solution of clustering the domains in a certain number to call the API. Here, the number of domains sent to the API for processing at a time was 50. This provided the results that were expected efficiently and accurately.

This operation is also highly dependent on the network and the load on the API server side. Also, it depends on the size of the blacklist or whitelist; normally, the blacklist is a bigger list to work on and takes more time to process, but with clustered calling it is faster. For example, a list of domains took 111 seconds to compute without batch processing and 76 seconds to compute with batch processing.

Table 4.3 provides experimental results on sample seed domains that show performance improvement after implementing the clustered API calls to filter out generated domains with existing blacklists and whitelists. The obtained results are based on 8 randomly selected blacklisted (seed) domains. Here, it can clearly be noticed how the old code didn't perform very well in scenarios with more than 1000 domains to process.

Table 4.3 Random experimental results after API cluster processing implementation

Domain	No. of Domains to process	Old Code (DPS)	Improved Code	% improvement
blogspot.com	6081	Couldn't process	529 s	100 %
camerquiz.com	553	66 s	16.6 s	74.8 %
judyguth.com	1553	147	46 s	68.7 %
chitraprakashan.com	1007	Couldn't process	127 s	100 %
drsridharspine.com	593	Couldn't process	56.6 s	100 %
0m66lx69dx.com	6081	Couldn't process	442 s	100 %
ankitbadigar.com	1072	132 s	73 s	44.7 %
hoteligeafiuggi.com	164	128 s	28.7 s	77.6 %
Average				**83.2 %**

4.5 Flask Application

To finish up this project, a Python-based Flask Web Application was created to host the code. Here, an HTML webpage is hosted locally using the Flask "render_template()" function, which has a textbox to input a domain to query. This webpage is rendered when a GET request is made. On the other hand, when a POST request is made, that is, by inputting a domain and submitting it, this will call the "mainRun()" function from the code and start processing the first part of the improved code.

The function then generates a JSON-formatted "payload" list with the domain and the outcome in it. The outcome is transformed into a JSON string using the json.dumps() method, which is then added to the "payload" list. To end, the function renders the output on the HTML page with the "payload" list.

Figures 4.1 and 4.2 show two screenshots of the Flask interface displaying the input and output domains for two different seed domains. Here, there are two sections of output where the first

part shows potentially malicious domains from the generated list and the second part shows already blacklisted domains from the generated output.

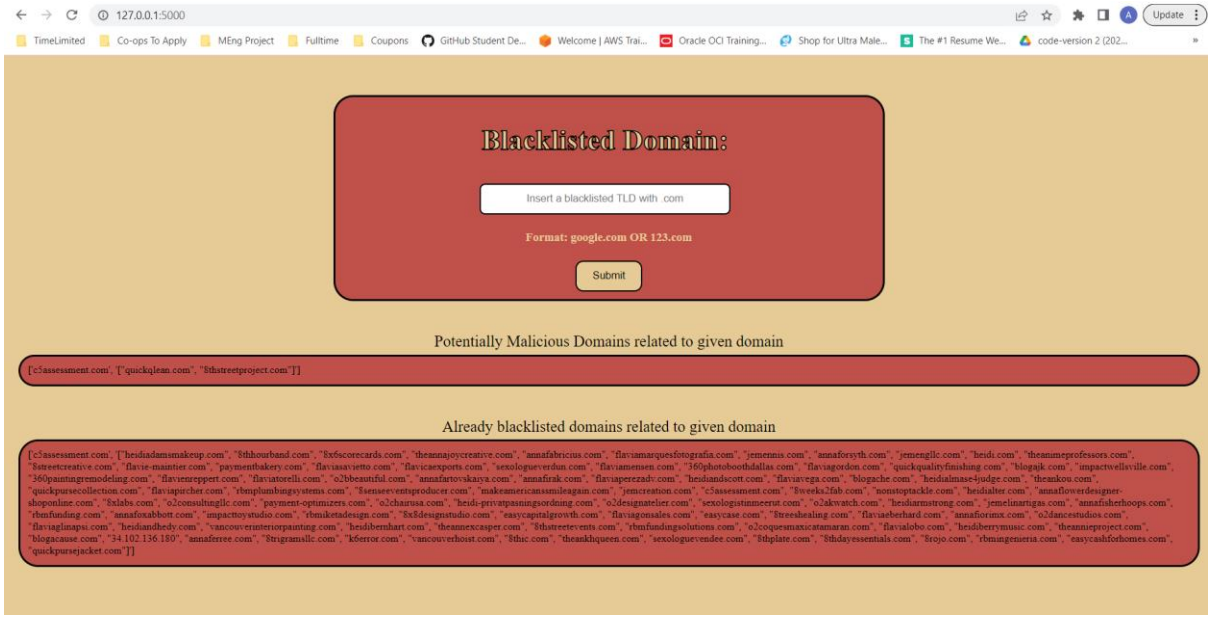

Figure 4.1 Flask App showing output domains for a sample input domain

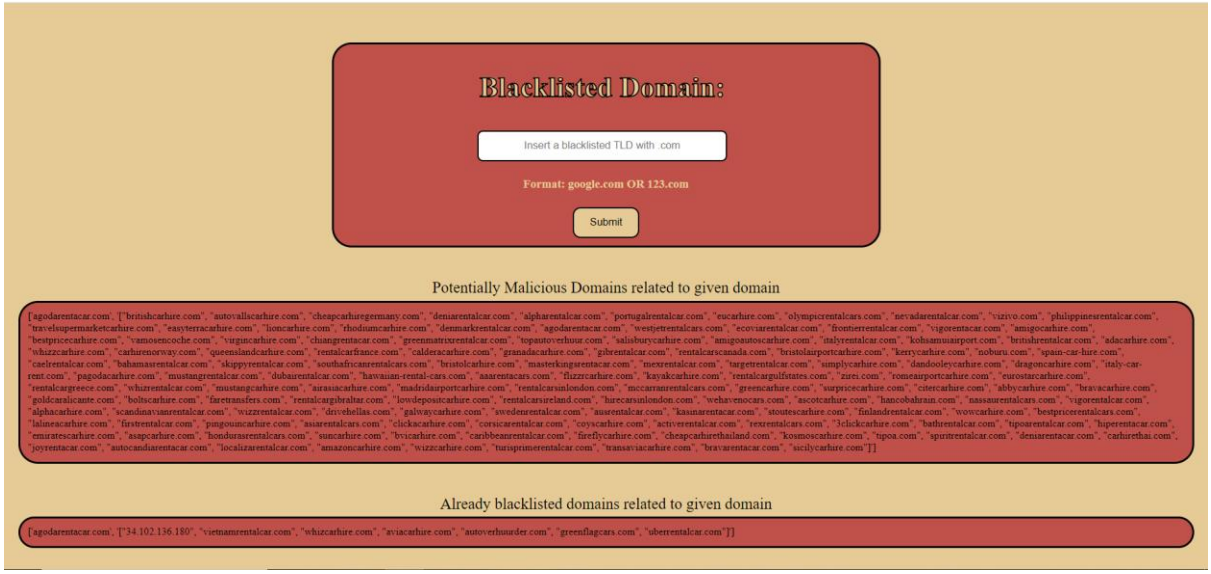

Figure 4.2 Flask App showing output domains for another sample input domain

4.6 Accuracy Testing

To test the accuracy of the output, outputs from 27 random malicious domains were concatenated and a dataset of 28176 domains was formed. Then, this information was checked against already available ISOT Lab domain classification services using deep learning and machine learning models. These algorithms were created to categorize domains as malicious or genuine depending on a variety of characteristics and features linked to them.

Upon querying the dataset, it was found that for all the domains provided to the system, the output was always classified as legitimate by the models. Thus, theoretically, it is good, as we are trying to predict the domains that might host malicious content in the future but are not blacklisted or malicious at the time of testing. However, it is hard to say that they will all be malicious in the future, and unfortunately, there is no such data available to verify the claims[1].

[1]A separate project is currently being conducted at the ISOT Lab focusing on evaluating the accuracy of the domain prediction system using machine learning models.

Chapter 5: Conclusions

With approximately 13 million malicious domains registered every month, based on a study done in September 2022, which accounts for over 20% of all newly registered domains, the threat of malicious domain-related activities keeps on increasing [8]. These malicious domains are used for several nefarious activities, including malware, adware, and phishing attacks. Although there are methods in place to identify and ban such domains, targeted attacks are still very difficult to stop. However, taking preventative measures can assist in lowering the likelihood of such attacks. The tested scheme, based on a malicious domain input, can provide a list of domains that may seem legitimate at the time of testing but have chances of being used for malicious activities in the near term. Early detection of these domains makes it easy to take the necessary precautions to stop their use for malicious ends. The domain prediction system processes the zone file provided and makes chunks out of it for faster processing. The domains related to the supplied domain are then collected and processed with WHOIS data. Once the list is available, it is filtered against existing blacklists and whitelists to provide the final output. Through implementation of these improvements, it was observed that there is a significant increase in the efficiency of the code. Specifically, we have achieved the following performance improvements:

A. Data management: 98.8 % improvement
B. Processing with Multi-processing: 85.7 %
C. Clustered API calls: 83.2 %

These improvements have resulted in a reduction in execution time, dependable outputs and improved overall functionality.

While the improved DPS generally produces dependable outputs, there may be instances where the outputs differ for the same input over time. This can occur when the blacklists and whitelists

used in the third part of the code to filter against existing lists are modified. These lists are regularly updated, which may result in varying results for the same input at different times. Furthermore, the system is designed to produce dependable output for malicious domain inputs that are blacklisted. Therefore, it is highly recommended to provide a suspicious domain for accurate output.

This system currently focuses on the ".com" TLDs, which make up over 50% of the market share, but it can be expanded to other TLDs as needed [7]. The future scope of this project can be to develop the same for other TLDs like “.net”, “.org” and others as per the ISOT industry partner’s needs. To use the code with a different zone file, the code just needs to replace the zone file, run chunking.py on the new zone file and it is ready to go. Further appending different types of zone files together would also work after making small changes to the first part of the code. Additionally, a dataset can be gathered and used for upcoming research using this system, which can be tried out to improve predictions with links to real-time malicious activity monitoring. In general, creating preventative measures like this scheme may assist in recognizing and minimizing the threat posed by malicious sites, thereby improving internet security. However, as hackers' strategies continue to change, it is essential to constantly enhance and upgrade such systems.

References

[1] ICANN Team. "Welcome to ICANN!" ICANN.org. https://www.icann.org/resources/pages/welcome-2012-02-25-en

[2] ICANN Team. "List of Top-Level Domains." Internet Corporation for Assigned Names and Numbers. https://www.icann.org/resources/pages/tlds-2012-02-25-en

[3] "concurrent.futures — Launching parallel tasks," The Python Software Foundation. https://docs.python.org/3/library/concurrent.futures.html#module-concurrent.futures.

[4] "Concurrency in Python - Pool of Processes." Tutorialspoint. https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_pool_of_processes.htm

[5] Ajay Goel. "Domain Blacklists – The comprehensive guide." GMass. https://www.gmass.co/blog/domain-blacklists-comprehensive-guide/

[6] Email Veritas. https://www.emailveritas.com/index

[7] Nadia. "How Many Domains Are There? (Ultimate Domain Name Stats 2022)." Siteefy. https://siteefy.com/how-many-domains-are-there/#total-number

[8] Stijin Tilborghs & Gregorio Ferreira, "Flagging 13 Million Malicious Domains in 1 Month with Newly Observed Domains," Akamai Security Research, 28 September 2022. https://www.akamai.com/blog/security-research/newly-observed-domains-discovered-13-million-malicious-domains

[9] Mark Felegyhazi, Christian Kreibich & Vern Paxson, 2010. "On the potential of proactive domain blacklisting". In *Proceedings of* 3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats https://www.usenix.org/conference/leet-10/potential-proactive-domain-blacklisting