Framework to Test Preparedness Against Quantum Computing

Authors: Atish Joottun**,** Poshan Peeroo, Kevin Yerkiah
Co-Author: Mr. Anwar Chutoo
atish.joottun@umail.uom.ac.mu, poshan.peeroo@umail.uom.ac.mu ,
kevin.yerkiah@umail.uom.ac.mu, a.chutoo@uom.ac.mu

Acknowledgements

We would like to thank Cyberstorm Mauritius (*CyberStorm – A group of Open-Source Developers, coding on the beautiful island of Mauritius. We are here to make the Internet secure.*, no date), and Mr Anwar Chutoo from the University of Mauritius (*University of Mauritius*, no date) for guiding, supporting, and helping us to make this project successful.

Abstract

This research test and analyses the websites that are hosted and available in Mauritius, to conclude whether Mauritius is prepared against Quantum Computing. This research can also be applied to any other countries. Our methodology includes the validation of ports and TLS servers, checking the presence of firewalls and finally, the actual test. All the code has been documented and uploaded on GitHub (*GitHub - AtishJoottun/Tldr_fail_testing: Testing Mauritius preparedness to Quantum Computing*, no date).

Introduction

In this fast-paced world where advancement of technology is exponential and the tools that we use is now extremely efficient and fast. One of such tools will be Quantum Computing, which theoretically has the potential to crack any classical encryption; “Rivest-Shamir-Adleman (RSA), Diffie-Hellman (DH), and Elliptic Curve (EC) Cryptosystems … relies on hardness of the Integer Factoring (IF) problems or Discrete Logarithm (DL) problem … which can be broken in polynomial time using Shor’s algorithm.” (Shaller, Zamir and Nojoumian, 2023).

The manner in which Quantum Computing operates, it takes advantage of quantum mechanics phenomena such as: Quantum superposition and quantum entanglement, making them much faster than classical computing methods. A quantum bit or qubit also known as a physical qubit is a basic unit of quantum information. Unlike classical bit, a qubit is a two-state quantum-mechanical system, meaning that a qubit has 2 bits, having 4 possible combinations at once. However, a qubit which is not in a perfectly isolated environment, will generate noise and errors. This noise will disrupt the information encoded in it.

That’s why logical quantum bits or logical qubits are used. Logical qubit is a cluster of physical qubits and within it, logical qubit quantum error correction will take place (What is Logical Qubit, 2024). Due to the quantum error correction, it provides a robust quantum computation even with the errors and noise. The number of logical qubits will determine how fast it can compute. The *Figure 1:Qubit Timeline* (*Quantum Computing Timeline by Gartner | Tmilinovic’s Blog*, no date) below shows how many logical qubits is required to crack some encryptions.

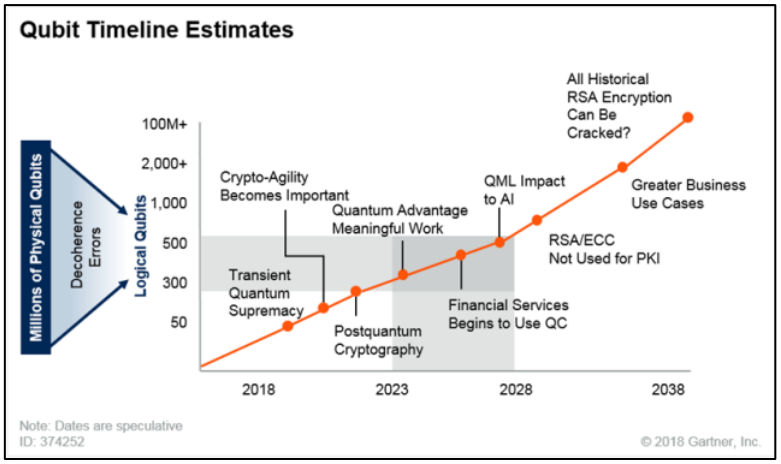


Figure 1: Logical qubits timeline estimates.

When powerful Quantum computing will be available, the IT infrastructure will be completely at risk. Due to the introduction of such tool, algorithms made to resist Quantum Computing were made. Algorithms like: *Kyber* or *NTRU* are known as Post-Quantum Cryptography or Quantum-Safe cryptography. The National Institute of Standards and Technology (NIST) (National Institute of Standards and Technology, no date), is amid of choosing a candidate to standardise the Post-Quantum Cryptography. Our Project will test the websites in Mauritius, which is on the public domain but anyone could replicate our methodology to check for their own countries or even a small network.

Using the script made by David Benjamin, we can check if the TLS of the website has been correctly implemented to support post-Quantum algorithms. More about his findings at tldr.fail (David Benjamin, no date).

What is the TLDR Bug?

Some misconfigured servers or the TLS middle boxes rejects the connections made with post-Quantum Secure Cryptography rather than negotiating for a classical cryptography.

Why does this happen?

A TLS *ClientHello* that uses post-Quantum encryption algorithms has a larger size than a TLS *ClientHello* that uses classical encryption algorithms. Thus, they exceed the packet transmission limit and needs to be send into multiple packets.

A buggy server can read the TLS *ClientHello* for the classical encryption as it is send in a single packet due to the fact that the TLS function *read()* is called once but it cannot read multiple packets of the *ClientHello* as it thinks that it's an error and terminates the connection.

Hence the reason for name **TLDR fail**. More information on David Benjamin's website: tldr.fail (David Benjamin, 2024).

How does the TLDR script works?

The script takes arguments the either the host name or the address using the argument *–addr <address>*. The script first establishes a connection with the host. Even if the host does not exist, it will still try to send a packet. This will result in an empty *ServerHello*.

When a connection is established, it will first send a large TLS *CilentHello* in a single write to the host server. If the host respond with a TLS *ServerHello* starting with "*b'\\x16\\x03\\ x03"* then the server passed the test. However, if the connection was rejected or the starting *SeverHello* was not as expected, the host failed the test.

After the first large TLS *ClientHello*, the script will then re-send the same large message in 2 writes. Again, if the output was abnormal or was not expected, it will consider the host as a fail.

The last test, it will send a smaller TLS *ClientHello* in the same manner as the larger *ClientHello*. It will first send the packet at once then it will send the packet in 2 writes.

If the host passes all the test, then the server was configured correctly, and post-Quantum cryptography could be implemented without any issue.

Getting the IPs for Mauritius

Each Country in the world has a range of IP addresses that they are allowed to use. They are assigned by The Internet Assigned Number Authority (IANA)(*Internet Assigned Numbers Authority*, no date), which is a standard organisation that oversees global IP addresses and other Internet-Protocol related to Internet Service Provider (ISP). Each country gets their IP from either the Local Internet Registry (LIR) or Regional Internet Registry (RIR). Mauritius' registry is AFRINIC(*AFRINIC the Region Internet Registry (RIR) for Africa - AFRINIC - Regional Internet Registry for Africa*, no date).

IP2Location (IP2Location, 2024) was used to identify IP address blocks assigned to Mauritius, totalling **6,482,688** addresses across **543 ranges**. Starting and ending IP addresses for each range, along with corresponding counts, were extracted into an Excel spreadsheet. A custom script shown below was then employed to generate a comprehensive list of individual IP addresses within these ranges by systematically incrementing each range based on its allocated count.

```
def extract_ip_addresses():
    range_of_ip_addresses = pd.read_excel(io='files/range_of_ip_addresses.xlsx',
sheet_name='ip_addresses')
    return range_of_ip_addresses[['start_ip', 'count']].values.tolist()

def get_ip_addresses():
    list = extract_ip_addresses()

    range_of_ip_addresses_with_count = list[3:4]
    ip_addresses = []

    for i in range(len(range_of_ip_addresses_with_count)):
        for j in range(range_of_ip_addresses_with_count[i][1]):
            ip_addresses.append(increment_ip(range_of_ip_addresses_with_count[i][0],
j))

    return ip_addresses
```

Scalable Data Processing

Recognizing the computational demands of processing a 6-million-entry IP address list and anticipating future large-scale datasets, we implemented a multiprocessing approach in Python for enhanced efficiency. Our custom script uses multiprocessing to distribute workloads across multiple processes. Flexibility is achieved by taking the *number_of_processes* as a parameter, which should divide evenly into the size of the data list. This script design also supports modularity, allowing users to define custom worker functions to match the specific processing needs of any list type.

```python
def start_multiprocessing(number_of_processes, worker, multiprocessing_return_function, *args):

    manager = multiprocessing.Manager()
    return_dictionary = manager.dict()
    jobs = []
    for i in range(number_of_processes):
        process = multiprocessing.Process(target=worker, args=(
            i, manager, number_of_processes, return_dictionary, args))
        jobs.append(process)
        process.start()

    for proc in jobs:
        proc.join()

    return multiprocessing_return_function(return_dictionary)

def default_multiprocessing_return_function(return_dictionary):
    return [item for sublist in return_dictionary.values() for item in sublist]
```

Validating the IPs

In the list of IP addresses, we need to filter those with port 443, which is HTTPS to check the TLS connection. For this task, Nmap Scripting Engine (NSE) was decided to be the appropriate choice as the tool was designed for such task. The simple NSE script checks if the IP address's port 443 is open. Below is the NSE Script.

```
portrule = function(host, port)
        if(port.state == "open") then
                if port.number == 443 then
                        print("open-443")
                else
                        print("open-80")
                end
        else
                print("closed")
        end
end

action = function (host, port)
        return nil
end
```

Using our previously established multiprocessing function, we integrated a Python worker function to execute the NSE script in parallel. IP addresses exhibiting open port 443 were logged into a CSV file, for further processing.

After validating the **6,482,688** IP Addresses, we got about **6963** that uses HTTPS/443. Then among the **6963**, we had to do more filtering to find how many were filtered/firewalled IP addresses.

Separating the Firewalled and Opened IP Addresses

To filter the IPs, we used the tool Nmap to see if we could ping the IPs to get the state of the port. In our case, it can be either filtered or open. We used this script below:

```
from utils.multiprocessing import start_multiprocessing, worker_check_if_behind_firewall, empty

with open('results/ip_addresses_with_port_open/ip_addresses_with_port80and443_open.csv', 'r')
as file:
   ip_addresses = [line.replace('\n', '') for line in file.read().splitlines()]

start_multiprocessing(129, worker_check_if_behind_firewall, empty, *ip_addresses)
```

The worker function *worker_check_if_behind_firewall* runs an Nmap command for every IP address assigned to the process. If the IP address was found to be "filtered", it was deemed to be behind a firewall or a middlebox. The filtered or open IP addresses are then separated into 2 different files: *ip_addresses_with_no_firewall.txt* and *ip_addresses_with_firewall.txt.*

This step significantly reduced the list from **6963** to **4779** IP addresses with potentially exposed TLS connections.

TLS server availability

After the separating the filtered IPs, we checked if the IP addresses have TLS servers in the port 443. To check this, we used *OpenSSL* to see if we could connect to the TLS Server. If connection is made, and we got ciphers, we could rule it out as TLS enabled. We separated each IP addresses into 2 files a shown in the code below.

```
def worker_check_tls_enabled(process_num, number_of_processes, *args):
   list_length = int(len(args[0])/number_of_processes)
   list = args[0][list_length*process_num:list_length*(process_num+1)]
   for i, ip_address in enumerate(list):
      tls_result = subprocess.run(["openssl", "s_client", "-connect", f"{ip_address}:443"], input=b'',
stderr=subprocess.PIPE, stdout=subprocess.PIPE)
      output = tls_result.stdout.decode()
      if "Cipher is (NONE)" in output:
         file = open("./results/tls_result/ip_addresses_with_tls_disabled.txt", "a")
         file.writelines(ip_address + "\n")
         file.close()
      else:
         file = open("./results/tls_result/ip_addresses_with_tls_enabled.txt", "a")
         file.writelines(ip_address + "\n")
         file.close()
      print(f"Process {process_num} is {(i/len(list))*100:.2f}% done.")
```

This worker function is called by another python file to execute the function on every instance of the IP Addresses. Through this process, we ultimately obtained a focused list of **3576** IP addresses potentially susceptible to TLDR bug which is stored at file *ip_addresses_with_tls_enabled.txt.*

Testing the IPs for TLDR BUG

Having identified a vast pool of potential Mauritian websites, our next step was to efficiently assess their vulnerability to the TLDR Bug. To automate this process and speed up analysis, we developed a script that integrated with David Benjamin's script. This solution iterated through the compiled list of open port 443 IP addresses that have been checked to not be behind firewall and have TLS enabled stored in the file *ip_addresses_with_tls_enabled.txt*, running the David Benjamin's script on every entry.

The resulting data was categorized as vulnerable or non-vulnerable and then exported to separate text files for further analysis. By making the use of multiprocessing within our script, we significantly reduced the overall processing time, enabling us to efficiently assess a substantial portion of the identified IPs. This is the worker script found in *multiprocessing.py*:

```
def worker_check_ip_addresses_tldr(process_num, manager, number_of_processes, return_dictionary, *args):
    list_length = int(len(args[0])/number_of_processes)
    list = args[0][list_length*process_num:list_length*(process_num+1)]

    for i, ip_address in enumerate(list):
        print(f"Process {process_num} is {(i/len(list))*100:.2f}% done.")
        try:
            result = subprocess.run(["python3", "./tldr_fail_test.py", ip_address], stdout=subprocess.PIPE, text=True,
timeout=240)
            if not "[WinError 10054]" in result.stdout:
                append_to_array(manager, str(process_num), (ip_address, False), return_dictionary)
            else:
                append_to_array(manager, str(process_num), (ip_address, True), return_dictionary)
        except Exception as e:
            print(e)
            append_to_array(manager, str(process_num), (ip_address, True), return_dictionary)
```

For every given IP address, the worker script will run the subprocess to call David Benjamin's script, tldr_fail_test.py. The output given by the subprocess is then analysed. If the error is produced, we append it to a dictionary.

This is code is found in *test_vulnerability.py*:

```
from utils.multiprocessing import start_multiprocessing, worker_check_ip_addresses_tldr,
default_multiprocessing_return_function

with open('results/tls_result/ip_addresses_with_tls_enabled.txt', 'r') as file:
    ip_addresses = [line.replace("\n", "") for line in file.readlines()]
file.close()
ip_addresses = start_multiprocessing(149, worker_check_ip_addresses_tldr, default_multiprocessing_return_function,
*ip_addresses)
vulnerable_ip_addresses = [ip_address for ip_address, failed in ip_addresses if failed]
non_vulnerable_ip_addresses = [ip_address for ip_address, failed in ip_addresses if not failed]

with open("./results/vulnerability_result/vulnerability_result.txt", "w") as file:
    file.write(f"Total number of ip addresses: {len(ip_addresses)}\n")
    file.write(f"Number of vulnerable ip addresses: {len(vulnerable_ip_addresses)}\n")
    file.write(f"Number of non vulnerable ip addresses: {len(non_vulnerable_ip_addresses)}\n")
    file.write(f"Percentage of vulnerable ip addresses: {(len(vulnerable_ip_addresses)/len(ip_addresses))*100}%\n")

with open("./results/vulnerability_result/vulnerable_ip_addresses.txt", "w") as file:
    file.writelines(ip + "\n" for ip in vulnerable_ip_addresses)
with open("./results/vulnerability_result/non_vulnerable_ip_addresses.txt", "w") as file:
    file.writelines(ip + "\n" for ip in non_vulnerable_ip_addresses)
```

Host Providers

To understand where the vulnerable websites were located, we used a script called "*whois-ip.nse*" from Nmap. This versatile tool retrieved the netname(*the name of the host provider*) associated with each IP address revealing the underlying network ownership or affiliation. With this crucial information, we further refined our analysis by classifying each vulnerable websites according to its corresponding netname. The extracted data was exported to two separate text files. The first provided a granular breakdown, grouping vulnerable websites by their respective netnames. The second offered a comprehensive overview, highlighting the distribution of netnames across the vulnerable IP addresses.

```
def worker_classify_ip_addresses(process_num, manager, number_of_processes, return_dictionary, *args):
    list_length = int(len(args[0])/number_of_processes)
    list = args[0][list_length*process_num:list_length*(process_num+1)]

    for i, ip_address in enumerate(list):
        print(f"Process {process_num} is {(i/len(list))*100:.2f}% done.")
        try:
            result = subprocess.run(["nmap", ip_address.strip(), "--script", "whois-ip.nse", "-Pn"], stdout=subprocess.PIPE,
text=True)
            netname = extract_netname(result.stdout)
            if netname == "":
                append_to_array(manager, "Unknown", ip_address, return_dictionary)
            else:
                append_to_array(manager, netname, ip_address, return_dictionary)
        except subprocess.TimeoutExpired:
            append_to_array(manager, "Unknown", ip_address, return_dictionary)
```

This is the function is called by the Multiprocess to create multiple instances for faster execution time. The *worker_classify_ip_addresses()* function is called in *test_vulnerability.py*. This returns the results of the web host providers that failed the test:

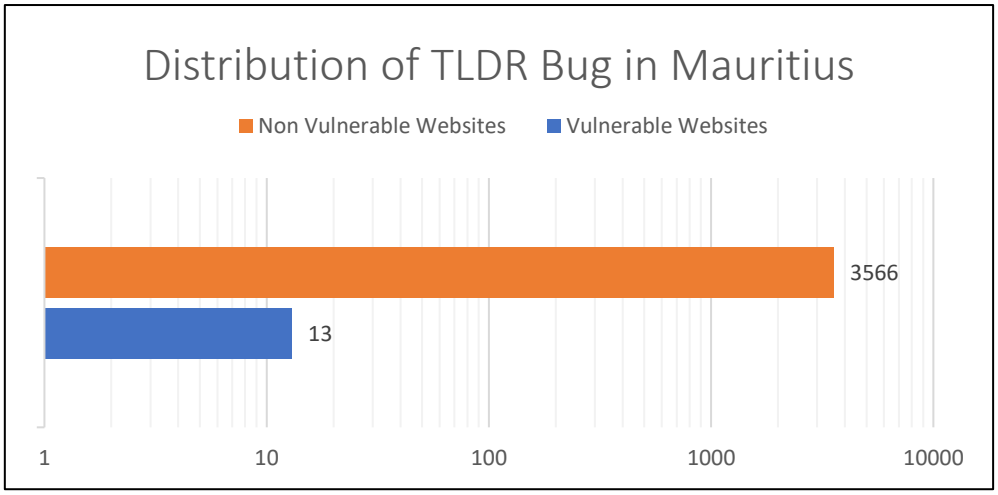
In our repository, we have a file named *classified_ip_addresses.txt* where we have listed all the IP which has errors corresponding to each netname.

```
with open('./results/vulnerability_result/vulnerable_ip_addresses.txt', 'r') as file:
    ip_addresses = [line.replace("\n", "") for line in file.readlines()]

dict_classified_ip_addresses = start_multiprocessing(13, worker_classify_ip_addresses,
classify_ip_addresses_return_function, *ip_addresses)

with open("./results/classified_by_provider/classified_ip_addresses_overview.txt", "w") as file:
    for key in dict_classified_ip_addresses:
        size_of_classified_ip_addresses = len(dict_classified_ip_addresses[key])
        size_of_all_ip_addresses = len(ip_addresses)
        percentage = (size_of_classified_ip_addresses/size_of_all_ip_addresses)*100
        file.write(f"{key}: {size_of_classified_ip_addresses} ({percentage:.2f}%)\n")

with open("./results/classified_by_provider/classified_ip_addresses.txt", "w") as file:
    for key, value in dict_classified_ip_addresses.items():
        file.write(f"{key}: {value}\n")
```

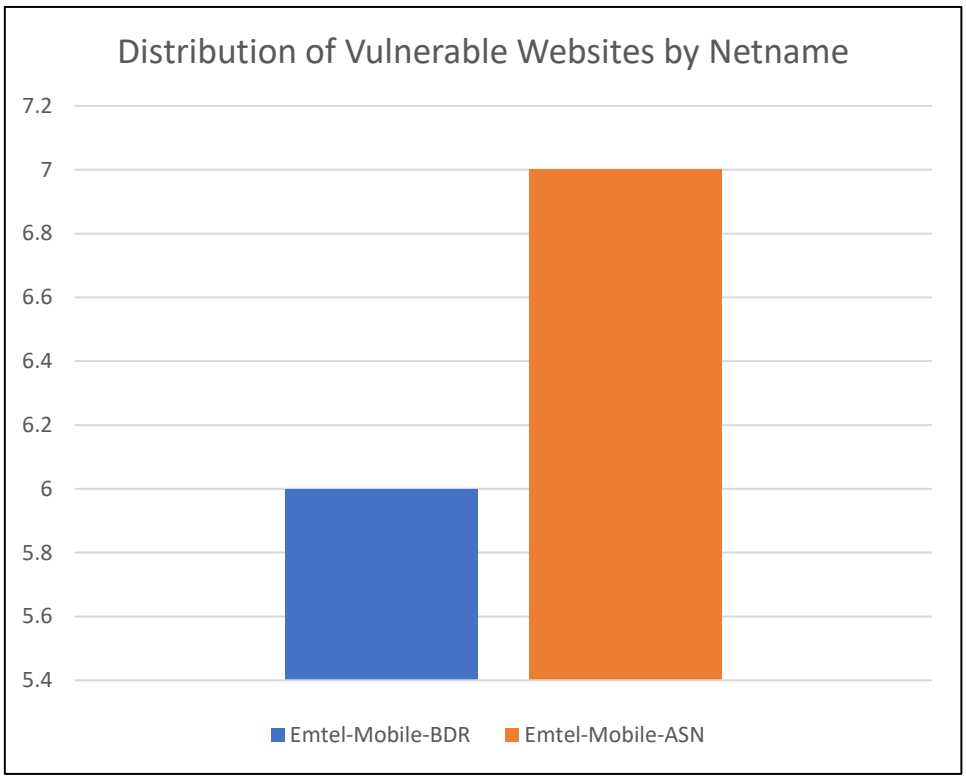
Data Analysis and representation

By employing these automated tools and meticulous data structuring, we were able to gain valuable insights into the vulnerability of Mauritian websites to the TLDR Bug, while also highlighting the potential scope and impact of this critical vulnerability within the region's digital ecosystem.

Vulnerability Prevalence



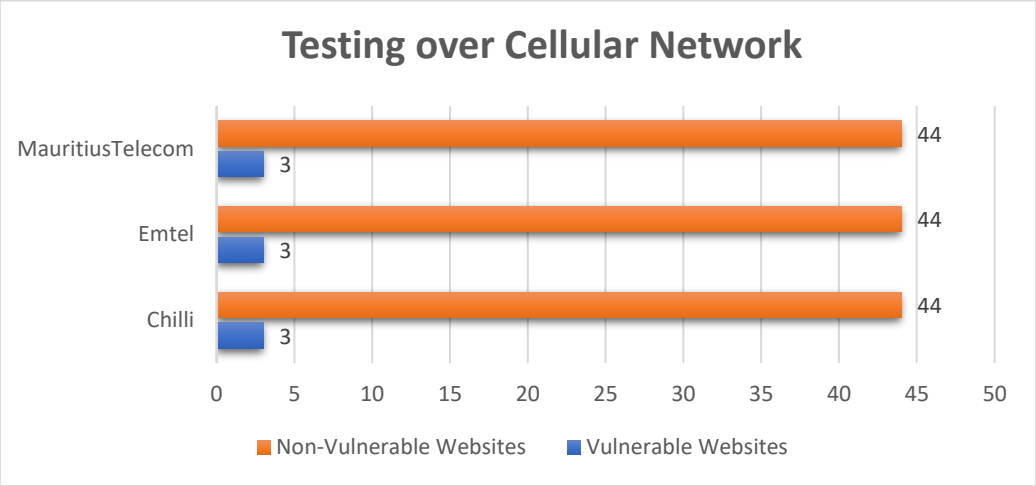
Netname Analysis (Special Case)



The IP addresses which have been found to fail against David Benjamin's script is found to be a special case. Where the TLS connects but does not return a *TLS ServerHello*. So, the actual failure rate is 0%.

TLDR Bug scanning over different Internet Service Providers

The bar chart below depicts the data analysed for the 47 most popular website in Mauritius. The data is the same for the 3 ISPs. Our data is consistent across all the ISPs.



However since **WI-FI** was used for this analysis, we might have some packet loss during transmission and due to limited resources we could only scan for these websites:
"google.mu, myt.mu, mcb.mu, okta.com, topfmradio.com, freshdesk.com, govmu.org, lexpress.mu, inside.news, intnet.mu, priceguru.mu, myjob.mu, mauritiustelecom.com, gceguide.com, vbazz.com, mauritiusturfclub.com, uom.ac.mu, lemauricien.com, france24.com, newsnow.co.uk, ib.mcb.mu, moodfeed.net, pixhost.to, journee-mondiale.com, frappe.cloud, partsouq.com, riddimsworld.com, fnb.co.za, naukrigulf.com, airmauritius.com, spikbuy.network, sfimg.com, mra.mu, mega.mu, instantly.ai, ifvod.tv, refinitiv.com, mycar.mu, canalplus.com, devskiller.com, Ebmu.sbmgroup.mu, Cyberstorm.mu, triobelisk.bandcamp.mu, outputmessage.bandcamp.mu, curtinmauritius.ac.mu, abcbanking.mu, radiologymauritius.mu."

Of these websites, only these 3: *"vbazz.com, mauritiusturfclub.com, moodfeed.net."*

These three websites failed due to expired domains. So, their names should be taken out of the top Mauritius visited websites.

Limitations

Due to restrictions of hardware, and lack of vantage point in Mauritius, we were not able to do a full scan on the most popular websites in Mauritius, at least top 300. We used laptops to run the programs which took lots of time to finish processing. We only scanned the IP Addresses in the WAN not on LANs, so we don't know the status of the internal IP addresses.

Improvements

One of the improvements that we could do is to get better resources. As for our vantage point, we could ask permission from companies to scan their internal network. This collaboration, will enable us to find the flaws that are hidden behind the TLS middle box or firewalls, thus ensuring a detailed scan of both the internal and external networks in Mauritius.

Conclusion

Our investigation reveals that 0% of the IP addresses that were tested failed against David Benjamin's script. The special cases were ignored since it does not return a TLS *ServerHello*. The 3566 IP addresses were found to be not susceptible to the bug. Thus, the TLS servers are deemed to be properly configured. Since, the failure rate is close to none, we can assume that Mauritius is a prime target to test post-Quantum encryption. Thus, Mauritius is well prepared against Quantum Computing.

Reference list

AFRINIC the Region Internet Registry (RIR) for Africa - AFRINIC - Regional Internet Registry for Africa (no date). Available at: https://www.afrinic.net/ (Accessed: 16 February 2024).

CyberStorm – A group of Open-Source Developers, coding on the beautiful island of Mauritius. We are here to make the Internet secure. (no date). Available at: https://cyberstorm.mu/ (Accessed: 16 February 2024).

David Benjamin (no date) Available at:https://*tldr.fail/ (Accessed: 16 February 2024)*.

GitHub - AtishJoottun/Tldr_fail_testing: Testing Mauritius preparedness to Quantum Computing (no date). Available at: https://github.com/AtishJoottun/Tldr_fail_testing (Accessed: 16 February 2024).

Internet Assigned Numbers Authority (no date). Available at: https://www.iana.org/ (Accessed: 16 February 2024).

IP2Location (no date) *https://lite.ip2location.com/mauritius-ip-address-ranges?lang=en_US*.

National Institute of Standards and Technology (no date). Available at: https://www.nist.gov/ (Accessed: 16 February 2024).

Quantum Computing Timeline by Gartner | Tmilinovic's Blog (no date). Available at: https://tmilinovic.wordpress.com/2019/01/18/quantum-computing-timeline-by-gartner/ (Accessed: 16 February 2024).

Shaller, A., Zamir, L. and Nojoumian, M. (2023) 'Roadmap of post-quantum cryptography standardization: Side-channel attacks and countermeasures', *Information and Computation*, 295, p. 105112. Available at: https://doi.org/10.1016/J.IC.2023.105112.

University of Mauritius (no date). Available at: https://www.uom.ac.mu/ (Accessed: 16 February 2024).

What is Logical Qubit (no date). Available at: https://www.quera.com/glossary/logical-qubit (Accessed: 16 February 2024).