Security Augmentations to the Delayed Fast Packet Algorithm
Project Final Report

Advait Erande, Andres Rodriguez, Wumian Wang, Yanzhou Yang

CS 9657A: Computer Networks II

Department of Computer Science
University Of Western Ontario

December 18,2020

Instructor: Anwar Haque

Abstract

The DNS is a system that allows the resolution of a URL to an IP address. As the technology is old, several security vulnerabilities exist in the system, which can quite easily be taken advantage of. One such vulnerability is DNS cache poisoning, which arises by way of the hierarchical structure of the DNS. An attacker can express their malicious responses into legitimate queries, and cause DoS at the minimum, and data stealing at worst. To execute this attack successfully, the attacker must push their response to the requesting DNS server before the authentic response. This paves the way for an approximation based on time the packet is received. A version of this approximation, the DFP, or delay fast packets, already exists, but is open to one crucial shortcoming: it is vulnerable to DoS attacks where, if the attacker repeats the attack for a virtually unlimited number of times, the user will never be able to access the requested resource. We address this vulnerability of DFP by modifying the way in which the DFP handles window conflicts. As a result, instead of having to wait for an undefined amount of time, we choose the most likely answer to be the correct one. Note that we do not claim to find a definitive answer to the question, just a more likely one. This project is based on the consideration that the existing DNS technology remains the same and is not changed in any way. We simply analyse the incoming packets on the client's side. Our results show that this framework was quite accurate in predicting the authentic and malicious packets.

Keywords: DNS, DNS cache poisoning, delay fast packets, DoS

Introduction

Domain Name System:

In the world of the internet, identification of a specific device or server anywhere in the world takes place by way of an Internet Protocol (IP) address. Whenever two devices communicate over the internet, they refer to each other by an IP address. An IP address is in the form of X.X.X.X, where X can range between 0 and 255, and can get more complex than that. So it is irrational to assume that someone may be able to remember the IP address for the websites or online resources that they wish to access. These addresses are thus given a name or a particular format to help remember how to access them. This is commonly known as the Uniform Resource Locator (URL). For example, instead of having to remember 172.217.164.206, one can simply remember "google.com" and by entering that into the browser window instead of the entire IP address, will get the user to the original website.

This is possible only by way of a program that translates the URL to an IP address. This program is the Domain Name System (DNS). It is not realistic for this translator program to reside in each computer, and perform the translation of every URL, since the number of URLs and the number of IP addresses that conform to this is virtually infinite. As a result, the DNS is in itself a server whose sole responsibility is this translation.

Structure of the DNS:

Because of the high number of requests and possible URLs, and for the sake of speed and redundancy, the DNS is not a single server located somewhere in the world. Instead, the DNS is a set of servers, organized in a tree, or hierarchical fashion. Let us observe the typical working of a DNS, in a simplistic form and a higher abstraction.

Suppose a client wants to know the IP address for a given URL. They send their request to the local DNS server (LNS), a server responsible for a small group of people and contains a set of the highly accessed and popular websites. Imagine that the URL request by the user is not a popular query, and the local DNS does not contain the IP address. Since the DNS is hierarchical, the local DNS server will query the next server (or the parent server) in the tree for the same query. If that server does not have the answer, it will request its parent, and so on. The DNS that does possess a valid answer will return the respective IP address to the requesting DNS, and the answer will trickle down back to the local DNS, and then the client that requested it. Each DNS server maintains its own cache of the translations and updates dynamically when new responses are obtained.

While this is a decent technique with respect to speed and accuracy, it does leave security holes. For example, if a higher DNS sends a response back to a lower level DNS, an attacker may try to intercept the packet and will modify the included IP address and change it to a malicious one in his control. The user, who will receive the malicious IP address response, will have no idea that that IP is not authentic, as he can only check the URL and not the IP address. As the local DNS server maintains a small cache of the recently and most used queries, this means the new infected IP address for the URL has now entered the cache. This is called DNS cache poisoning. And for future queries, the local DNS may respond with the infected IP address.

Obviously, this can prove to be a major problem, where many clients may get infected and lose personal data, if not worse. Clearly, this has to be mitigated in some form. The techniques we will observe today must use the DNS as it exists today and cannot change or modify the system or packets and their formats in any way. We must make use of whatever technologies and interaction we have available.
For mitigation against these types of attacks, one proposed solution is the Delay Fast Packet (DFP) algorithm. In the case where we have two responses, without any other protection mechanism, the first packet received will be promptly put to use and the second packet will be discarded. As a result, from the attacker's perspective, it is crucial to get their packet in before the valid packet does. Hence, more often than not, the attacker's response will be first. This, however, is not a norm and does naturally have exceptions. Since we receive more responses from the parent DNS than we do from the attacker, we can build a statistical prediction about the DNS responses, and use it to identify the malicious ones from the attacker. However, the DFP algorithm has a crucial vulnerability: it has the propensity to lead to Denial of Service (DoS) attacks. We thus introduced a modified framework for mitigating against the DoS attack, and propose it as a viable solution.

Section II will cover a basic introduction into the DFP algorithm [1] and its shortcomings; Section III will look into our proposed solution; Section IV will take a look into the related works; Section V deals with the implementation of our framework and Section VI will look at the results and the analysis of those results; and Section VII will look and future scope and improvements.

DFP Algorithm Summary

The paper 'Delay Fast Packets (DFP): Prevention of DNS cache poisoning' by David, Dolev and Anker presents an algorithm to detect and prevent attempts of cache poisoning attacks [1]. In order for a successful cache poisoning attack to occur two conditions must be met, the malicious packet must contain the transaction ID that corresponds to the original request and the malicious packet must arrive before the authenticate response packet. In this paper the assumption is made that an attacker is eavesdropping, which results in them being able to always generate an acceptable malicious packet. In order for the malicious packet to arrive before the authenticate packet the attacker will inherently send their response as soon as possible, as a result the malicious packet will arrive significantly faster than authenticated response from the authoritative server (ANS).

The authors contributions are two-fold. They present a strict model that is able to prevent attacks from a powerful adversary. To their knowledge, they are the first to introduce a security engine that does not change the DNS protocol and yet protects against eavesdropping attacks which are able to generate valid malicious packets. Given that DNS request/response packets are unencrypted and are broadcast to anyone who chooses to listen, anybody with access to the copper infrastructure can eavesdrop on the transmissions. Additionally, the proposed solution can be implemented as a black box. No modifications to the DNS protocol nor to the BIND (Berkeley Internet Name Domain) server code are required.
The primary goal for the DFP algorithm is for estimating the RTT between the DNS server and each of the authoritative servers and to delay the responses that arrive too fast. The algorithm takes into consideration the different service types (MX, A, AAAA, CNAME, etc), given that they will vary in processing time. The algorithm estimates a RTT needed for the next response to arrive from each authoritative server and service type. In the scenario where a valid packet arrives prior to the predicted RTTT, the algorithm will wait a certain amount of time before accepting and forwarding the response. If however another valid packet corresponding to the same request arrives during the time window, both response packets are dropped and a new request with the original request data is generated. This however exposes the algorithm to a potential DoS (Denial of Service) attack. If an attacker is able to consistently send a response packet for each request then the DFP algorithm will continuously drop both packets resulting in the user never getting a response. A high level diagram is presented in Figure 1, depicting a cache poisoning attack from an eavesdropper. Figure 2 depicts the proposed DFP algorithm defending against a cache poisoning attack.

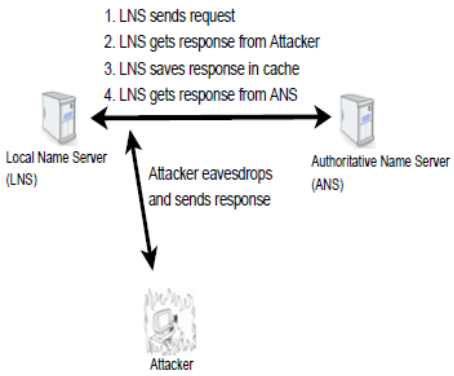

Fig. 1. Cache Poisoning Example

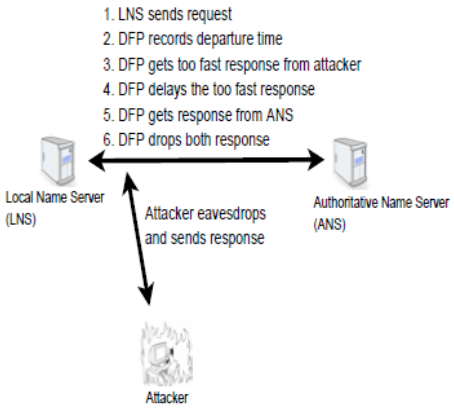

Fig. 2. DFP Operation

The DFP algorithm implements two hashtables, PacketDictionary maps the outgoing requests with their corresponding incoming responses. StatsDictionary stores the statistics for each authoritative DNS server. From each packet that arrives a key is constructed using the authoritative server IP, transaction ID, and packet type. If the packet is a request, it will be saved in the PacketDictionary hash table. If the packet is a response, its corresponding DNS request is retrieved using its key. In the scenario where there is no matching request, the received packet will simply be dropped. In the scenario where two receive packets have arrived for a single request, the algorithm will remove the request and both response packets from the PacketDictionary. In the regular scenario where a request packet receives a single response packet, the algorithm will calculate the RTT between the local DNS server (LNS) and authoritative server (ANS). Additionally it will calculate the EstimatedRTT, DevRTT and estimates the window time frame.

The DFP algorithm uses the following formula to detect packets arriving too fast:

$$RTT < EstimatedRTT - DevRTT * FactorWindow$$

The two parameters influencing the EstimatedRTT and DevRTT are alpha and beta , they determine the weight of the new RTT sample against the history which influences the window starting point. Through experimentations using real DNS traffic data it was determined that since the deviation of the RTT is relatively low. Thus the values for and were set as 0.125 and 0.25 respectively. The main considerations for FactorWindow is the tradeoff between the number of false positives and the probability of a successful attack. Through experimentations it was determined that a value of 2 provides the best results.

The resulting algorithm is capable of preventing cache poisoning attacks, however the algorithm presents a weak spot with its vulnerability to DoS attacks.

Proposed Solution

Currently the DFP algorithm uses packet data to determine a time window in which a valid packet should arrive. If a packet arrives before the window, it will delay the packet and wait to see if any packet arrives in the window. If another packet does arrive, it discards both and queries the DNS server once again. This creates an issue where an attacker can eavesdrop and constantly send malicious packets, given the DFP algorithm it will discard both received packets and query the DNS server again. As a result, an attacker can easily generate a DoS attack.

The proposed solution is a security augmentation to the current DFP algorithm. In case of multiple received responses, a database will be created to keep track of each response packet, along with the information of whether it arrived inside the window or not, and to which request packet it corresponds to. Then, it will generate an identical request to the original and send the request again, under the assumption of an eavesdropping attack it will expect two responses, which will then be inserted into the table again. Since the attacker will falter in judgment of the time, this will inherently lead to the malicious response to arrive outside the window more often and the authentic response inside the window more often. Since the database keeps track of the received packets, the success rates of each occurrence will be used to predict the authentic response. Mininet will be used to implement the DNS servers and simulate the attacker eavesdropping and sending malicious packets. In order to determine which packet is authentic the algorithm will begin by hardcoding the algorithm to iterate an additional five times per original request. Additional experiments will be conducted using varying iterations in order to determine an efficient number of iterations required for determining the authentic packet. Our proposed solution aims to alleviate the shortcomings of the DFP algorithm's inability to handle potential DoS attacks. The solution comes at a trade-off, a higher security will result in a greater latency between the initial request and the authenticated response.

Related Works

Several previous studies aim to detect and/or mitigate cache poisoning attacks are described as follows. Domain Name Security Extension(DNSSEC) is a protocol defined in RFC 4033 and its predecessors[2]. This protocol aims to provide data authenticity by asymmetric key encryption. However, previous work indicates that DNSSEC introduces new issues, including key management, time synchronization and notable computational overhead[3]. Also it is challenging to use DNSSEC to prevent cache poisoning attacks because it requires deploying new servers or reinstalling the protocol in the existing servers.

Chau et al. proposed an adaptive deterrence framework called CGuard, which passively monitors upcoming DNS packets[3]. When CGuard detects an abnormal number of DNS responses within a certain time period, it switches to a variety of high-confidence channels, such as DNSSEC or DNS over TLS, and uses these channels to accomplish DNS lookup[4].

In the work of Y. Jin et al, they used machine learning techniques[5]. Some of the features used to train the model include the 5-tuple information of a DNS packet, the updated GMT based time, geographic information of new IP address, distance between the new and old IP address, distance between the new and old domain name of NS records[5]. Y. Jin and his team tried a different approach to address the problem of detecting the cache poisoning attacks than us, however, their solution is much more expensive than ours, since they need a huge training data set for their model, and they do not provide a way to minimize the effects of attack after detecting the attack.

A.Herzberg and H.Shulman gave another solution to detect and prevent the cache poisoning attack, which is “The Sandwich antidote to DNS poisoning”, their logic is that, the sandwich antidote will hold a table that stores all outbound DNS requests, and it also keeps track all of the inbound DNS responses, and if the sandwich antidote detects that the validation fields of one packet, for example the transaction ID, does not match the corresponding value that stored in the pending DNS request, it will then switch to the prevention mode, in which case, the antidote will keep sending 2 invalid DNS requests, and one normal DNS request to the server, until it gets the 3 expected responses in order[6]. Their solution did provide a highly secured way to detect the cache poisoning attack, and unlike the machine learning solution provided by Y. Jin et al, the sandwich antidote algorithm would also provides a way to mitigate the effect of the attack similar to ours : by keep sending the request to the server until they get the expected the response.However, the sandwich antidote requires a large table that can store the information of all of the outbound DNS requests, and also it needs to validate the validation fields of each of the packets, the performance cost of such an algorithm would be undoubtedly higher than our solution.

Implementation

Mininet Setup/ Network Topology

To simulate a DNS cache poisoning attack, the first step is to build a network topology using Mininet and a Floodlight remote controller. Mininet allows users to build virtual networks on a virtual machine, and supports running the Java program on different hosts. the simulated network topology is described as follows.

Figure 3: Simulated Network Topology.

The virtual network has three hosts (h1, h2 and h3) and five switches (s1, s2, s3, s4, s5). Host h1 represents a DNS client, a.k.a, a local DNS resolver that sends a DNS query to the DNS authoritative server. Host h3 represents an authoritative DNS server, when it receives a DNS query, it generates a response message that contains the IP address to the query domain name, and sends the response back to the client. Host h2 represents the attacker. When h2 observes a DNS query is sent to the server, h2 will generate a malicious response message and send it back to the client. In the network topology, h2 is located between the DNS client and the DNS server to simulate an eavesdropping attacker. Links between a host and a switch (h1-s1, h2-s2, h3-s5) have 15Mbps bandwidth, 1ms latency, no packet loss. Links between two switches (s1-s2, s2-s3, s3-s4, s4-s5) have 15Mbps bandwidth, 10ms latency. The packet loss rates are set as either 0 or 2% for links between switches, depending on the experiment conditions.

Basic DNS System

The DNS simulation system consists of three major components: the DNS message encode/decode framework, a simple DNS client, and a simple DNS server (Github reference: https://github.com/wumianwwm/ dns_java_simulation). For the client, the framework provides API so that for every domain name, the client can create a DNS query message, which has the same format defined in RFC 1035, with one DNS question. For server or attacker, the framework allows them to generate a DNS response based on a received query. To simplify our implementation, every response message has only one type A resource record, which contains an IP address to the query domain name, in the Answer section. No resource records are added in Authority or Additional sections. The encode/decode framework also provides an API for serialization/deserialization, which allows for converting a DNS message to a byte array, or extracting a DNS message from a byte array in a datagram packet.

When the DNS server receives a query, it creates a response with “192.127.112.31” as answer to the query’s domain name. Then the server has 80% possibility to wait for 50ms, then sends response to client; otherwise it waits (50 - X)ms, where 10 <= X < 45. Combined with mininet topology, the estimated

Round Trip Time (RTT) between client h1 and server h3 is close to 130-140ms. When the DNS attacker receives a query, it creates a response with “101.102.201.202” as answer to the query’s domain name. Then the attacker has 80% possibility to immediately send this malicious response to client, otherwise, it delays 95 - 125ms before sending the response. The above settings allow us to simulate changes in network condition, so that we observed several edge cases in experiments. For example, an attacker's packet arrived later than the server's packet; after the client received one packet, the other arrived too late that it passed the client wait time, so client took it as only one response to one query, and took that packet as a valid packet.

The client runs in sequential mode, which means every time it looks up one domain name, and it does not process another domain name until it determines IP address of the current domain name, or fails to do so. To simulate eavesdropping attack, for each lookup domain name, the client sends two identical queries, one to the attacker and one to the server. However, when client receives a packet, it can only use the packet's RTT, and our improved algorithm, to determine whether the packet is valid or malicious.

Improved Algorithms for Encountering Multiple Responses to One Query:

sendAndRecv_oneQuery(ServerStats, queryName):

1. *createAndSendPacket(queryName)*
2. firstReceivedPacket <- *receive()*
3. waitTime <- ServerStats.*calculateWaitTime(firstReceivedPacket.RTT)*
4. secondReceivedPacket <- *receive(timeout = waitTime)*
5. **IF** (secondReceivedPacket != NULL) && (*both_Packet_Have_Valid_ID_QueryName()*)
6. && (*both_Packet_From_Different_Sources()*))
7. PacketStatsArray <- *createPacketStatisics(firstReceivedPacket, secondReceivedPacket)*
8. **LOOP (Max 5 times):**
9. receivedPackets <- Repeat_Step_1_to_Step_4
10. **FOR** packet **IN** receivedPackets:
11. **IF** firstPacketStats.*inChargeOf(packet):*
12. firstPacketStats.*updateInWindowTimeCount(ServerStats, packet.RTT)*
13. **ELSE IF** secondPacketStats.*inChargeOf*(packet):
14. secondPacketStats.*updateInWindowTimeCount*(ServerStats, packet.RTT)
15. **END IF**
16. **END FOR**

```
17.     END LOOP
18.     IF firstPacketStats.InWindowTimeCount > secondPacketStats.InWindowTimeCount
19.         RETURN firstReceivedPacket
20.     ELSE IF secondPacketStats.InWindowTimeCount > firstPacketStats.InWindowTimeCount
21.         RETURN secondReceivedPacket
22.     ELSE RETURN NULL // can't distinguish which packet is valid one.
23.     END IF
24. END IF
25. RETURN firstReceivedPacket // Only one packet with matched ID/query name, return it.
```

Above is a simplified version of pseudo code for the improved DFP algorithm. To obtain estimatedRTT and deviation RTT (devRTT) between the client and server, the algorithm first creates server statistics by sending packets to server only, and records responses received time to calculate estimatedRTT and devRTT. After sending a query, the client waits for the first response. For simplicity, handling loss of first receiving packet is omitted in above pseudo code. We use a similar method as the original DFP paper to determine whether a packet arrives too early, i.e., before window start time, and use the same formula to calculate wait time for early arrived packets. If a packet's RTT is greater than window start time, i.e., not an early packet. The client still waits for 2 * devRTT to see if another packet arrives.

In the implementation, after receiving another packet, the algorithm checks whether they have the same source IP address, valid ID and query name. Handling edge cases like both packets come from the same IP address, is implemented but omitted in pseudo code. If both packets have matching IDs, same domain name as the query's, from different sources. The client creates packet statistics for both one of them. PacketStatistics contain source IP, query domain name, IP to domain name, and count how many times the packet arrives in the window. Then the client resends the query, for every response the client receives it calculates the wait time if it needs to wait another packet, and compares received packet with packetStatistics. If one packet comes from the same source, has the same query domain name as a packetStatistic, it is in charge of that packet. The packetStatistic then checks whether the received packet arrives too early, if the packet RTT is greater than window start time, we increment packetStatistcs.InWindowTimeCount by 1. After resending is completed, the algorithm compares two PacketStatistics InWindowTime counts, the one with higher counts means that statistics corresponding to the packet come from the server. Update server statistics (ServerStats) using verified, valid packets, is implemented but omitted from above algorithm.

Mininet Limitations & Implementation Issues

The mininet script is written in Python, but our DNS system is written in Java. Therefore, it is not possible to control the behaviour of switches or the remote controller. In addition, sometimes the mininet does not simulate link latency properly. Occasionally, it is observed that the estimated RTT between h1 and h3 was much less than 100ms, when it was expected to be within 130-140ms. Data collected in those rounds of experiments were considered as suspicious and were not used in data analysis.

Data Analysis

In order to quantify the effectiveness of the Improved DFP algorithm, the original DFP algorithm must first be analyzed to further understand its limitations.

In the event of a powerful adversary that is able to constantly send malicious packets through eavesdropping, the original DFP will falter given its vulnerability to DoS attacks. There are two scenarios in which the attacker will be successful in their cache poisoning and one scenario for a DoS attack. The first scenario in which a cache poisoning attack is successful, is if the adversary is able to successfully send the malicious packet such that it arrives inside the DFP window and before the authentic packet. In this case the DFP algorithm will automatically accept the malicious packet. The second scenario is if the authentic packet arrives after the DFP window, in this case it does not matter whether or not the malicious packet arrives before the window or in the window. Since the authentic packet arrived outside the window, the malicious packet will automatically be accepted. In the case of DoS attack, given that the adversary has the ability to detect all outgoing DNS requests they will always be able to generate a valid malicious packet. Since the DFP algorithm will continuously discard the request and re-query when two packets are received, so long the attacker's malicious packet arrives before the window a DoS will occur. Given that the adversary is located closer to the LNS, relative to the ANS, the malicious packet will almost always arrive early.

Before looking at the data, a quick overview of how the improved algorithm alleviates the current limitations of the DFP algorithm. In the original DFP algorithm if an adversary sent a malicious packet such that it arrives inside the window but before the authentic packet, the malicious packet would automatically be accepted. In the improved algorithm, regardless of which packet arrives first the algorithm will wait the entirety of the window to ensure the authentic packet arrives. In the case of a DoS attack, originally if an adversary would continuously send malicious packets the DFP algorithm would result in a DoS event. However with the improved algorithm the continuous malicious packets are used to

determine which of the two packets received is the authentic, thus regardless of continuous packets being sent the algorithm will after a certain number of iterations be able to determine the authentic packet.

For the baseline experiment the parameters were set such that the rescue method would iterate one to five times individually. Each iteration variation was tested on 100 DNS requests. Table 1 shows the data collected, for each experiment the rate at which authentic packets and malicious packets are shown. As well as the rate in which the algorithm was inconclusive and the latency between the original request and the final response determined by the improved DFP algorithm.

Rescue Iterations	Authentic Packet	Malicious Packet	Inconclusive	Latency
1x	74%	4%	12%	311.47ms
2x	91%	3%	6%	457.55ms
3x	97%	1%	2%	577.38ms
4x	99%	0%	1%	739.73ms
5x	97%	3%	0%	890.24ms

Table 1. Baseline experiment results

From the data collected it can be seen how additional iterations of the algorithm improves the success rate of authentic packets identification and reduces false positive rate. Looking at the false positive data, it is shown that with additional iterations the rate of false positives decreases except between the four and five iterations. In order for the algorithm to falsely accept a malicious packet it would require the malicious packet to arrive inside the window more times than the authentic packet. Given that the adversary is located closer to the LNS than the ANS is to the LNS, it is more likely for the malicious packet to arrive before the window. However considering the network's latency variability it is possible for the malicious packet to arrive in the window and for the authenticate packet to arrive before the window. While also taking into consideration the small sample size of the experiment, this explains the decrease in accuracy of the algorithm. A larger sample size would provide with more accurate results, nonetheless the experiment shows the general trends for improved accuracy with additional algorithm iterations.

Inconclusive results occur when the algorithm is unable to determine which of the two packets received is malicious. When two packets are received the algorithm will store its information alongside whether it arrived inside or outside the window. When the algorithm only runs an additional iteration, it has a larger chance of producing inconclusive results. Generally a malicious packet will arrive before the window, but due to network variability it is possible for the packet to arrive inside the window. As a result with less algorithm iterations the probability of a malicious packet to arrive inside the window at the same rate as an authentic packet is much greater than when the algorithm runs the algorithm more times. This can be seen as the rate of inconclusive results goes from 12% all the way down to 0% as additional algorithm iterations occur. Additionally it is shown with additional algorithm iterations the latency will naturally increase. In the case of a single additional iteration the total latency is 311.47ms while the latency for five additional iterations is 890.24ms. Although additional iterations result in greater latency it is worth noting, whenever the algorithm results in inconclusivity if the client wants to receive a response the DNS will be required to resend the original response and thus running the algorithm additional times. Meanwhile with five additional iterations, although the latency is significantly greater it will result in a conclusive response with a high degree of certainty that the response received will be authentic.

Although the initial experiment results showed effectiveness towards preventing a DoS attack, the network simulations do not mimic a real world network. In order to get a better understanding of the algorithm's effectiveness in a more realistic scenario, the simulation was run with the addition of a 2% packet loss in each link. Table 2 shows the results from the more realistic network simulation.

Rescue Iterations	Authentic Packet	Malicious Packet	Inconclusive	Latency	Packet Loss Parameter
1x	73%	15%	12%	451.49ms	2%
2x	83%	15%	2%	645.69ms	2%
3x	86%	10%	4%	905.4ms	2%
4x	82%	14%	4%	1083.4ms	2%
5x	80%	20%	0%	1301.1ms	2%

Table 2. Experiment results with the additional parameter of a 2% packet loss per link.

Looking at the success rate of the algorithm there are a couple points worth noting. First the significant decrease in performance, the baseline experiment's best parameter setting achieved a success rate of 99%. However in the more realistic network simulation the highest success rate achieved was a mere 86%. Looking at the simulation's network topology in Figure 3, the number of links a packet has to travel through in order to reach the client is double the number of links the malicious packet has to travel. As a result the authentic packets have a greater chance of being lost than the malicious packet. This is a plausible explanation to the decrease in performance of the algorithm.

Secondly, the algorithm achieved its best success rate of 86% using only three additional algorithm iterations. Meanwhile the addition of a fourth and fifth iteration resulted in a decrease in the algorithm's ability to successfully detect the authentic packet. This due to the packet loss that was introduced, since authentic packets have double the chance of being lost in transmission the more times the algorithm re-queries the more likely an authenticate packet will be lost. Which will negatively affect the algorithms ability to determine which of the two packets is malicious and which is authentic.

With the addition of packet loss in the network the algorithm increased in latency roughly 50% over the baseline values. During the algorithm if a packet is lost, the algorithm will unknowingly wait the entirety of the window waiting for the second packet to arrive. As a result the latency values are significantly increased.

Future Work

Currently our algorithm and simulation are implemented in such a way that it will only handle sequential requests. In order to obtain more accurate data that represents its effectiveness in the real world, the algorithm and simulation must be modified such that they are capable of handling multiple requests at the same time. Additional experimentations will be required to determine the memory cost of maintaining multiple requests simultaneously. In the experiments when packet loss was introduced, the latency values increased dramatically. Future research would include modifications to the algorithm to predict potential packet loss occurrences, in doing so will reduce the issue of the algorithm unknowingly waiting the entirety of the window when a packet has been lost. During the experiments it was discovered that by varying the algorithm’s iterations the level of security could be controlled but with a trade off in latency. Additional research could involve using machine learning techniques to adjust the algorithm’s security level depending on different periods of higher risk.

Conclusion

This paper presents an improvement to the DFP algorithm that addresses its vulnerability to DoS attacks. The proposed algorithm takes advantage of the high probability of malicious packets arriving before the window to identify the malicious and authentic packet. Considering the algorithm makes no modifications to the current DNS protocol, it can be implemented as a black box at a low deployment cost. Through simulations the improved algorithm was shown to be capable of detecting and preventing DoS attacks with a high degree of success. Given the algorithm's high flexibility, the degree of security can be adjusted to accommodate for the latency requirements. The major trade off between a high degree of security is the relatively large latency. Future work will focus on implementing a smart dynamic algorithm that is capable of adjusting its parameters to reduce latency during periods of safe network transmissions.

References

[1] Tzur-David, S., Lashchiver, K., Dolev, D., & Anker, T. (2011, September). Delay fast packets (dfp): Prevention of dns cache poisoning. In *International Conference on Security and Privacy in Communication Systems* (pp. 303-318). Springer, Berlin, Heidelberg.

[2]Arends, et al. DNS Security Introduction and Requirements https://tools.ietf.org/html/rfc4033

[3] S. Ariyapperuma and C. J. Mitchell, "Security vulnerabilities in DNS and DNSSEC," *The Second International Conference on Availability, Reliability and Security (ARES'07), Vienna*, 2007. pp. 335-342.

[4] S. Y. Chau, O. Chowdhury, V. Gonsalves, H. Ge, W. Yang, S. Fahmy, and N. Li, "Adaptive Deterrence of DNS Cache Poisoning," *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering Security and Privacy in Communication Networks*, 2018. pp. 171–191,

[5] Y. Jin, M. Tomoishi and S. Matsuura, "A Detection Method Against DNS Cache Poisoning Attacks Using Machine Learning Techniques: Work in Progress," *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*, Cambridge, MA, USA, 2019, pp. 1-3, doi: 10.1109/NCA.2019.8935025.

[6] S. Herzberg and A.Herzberg , "Unilateral Antidotes to DNS Cache Poisoning," Sep. 2012.