Delay Fast Packets (DFP): Prevention of DNS Cache Poisoning

Shimrit Tzur-David Kiril Lashchiver Danny Dolev Tal Anker

School of Computer Science
The Hebrew University Of Jerusalem
shimritd,kiril,dolev,anker@cs.huji.ac.il

Abstract. The Domain Name System (DNS) protocol is used as a naming system for computers, services, or any other network resource. This paper presents a solution for the cache poisoning attack in which the attacker inserts incorrect data into the DNS cache. In order to successfully poison the cache, the attacker response must beat the real response in the race back to the local DNS server. In our model, we assume an eavesdropping attacker that can construct a response that is identical to the legal response. The primary aim of our solution is to construct a normal profile of the round trip time from when the request is sent until the arrival of the response, and then to search for anomalies of the constructed profile.
In order to poison the cache of a DNS server, the attacker has to know the source port and the Transaction ID (TID) of the request. As far as we know, all current solutions which do not change the protocol, assume an attacker that cannot see the request and therefore has to *guess* the TID. All these solutions try to increase entropy in order to make the guesswork harder. In our strict model, increasing entropy is useless. We in no way claim that our scheme is flawless. Nevertheless, this effort represents the first step towards preserving the DNS cache assuming an eavesdropping attacker.

1 Introduction

The Domain Name System (DNS) [1], [2] is a hierarchical naming system built on a distributed database for computers, services, or any resource connected to the Internet or a private network. The DNS distributes the responsibility of assigning domain names and mapping those names to IP addresses by designating authoritative name servers for each domain. Authoritative name servers are responsible for the domains in their jurisdiction. In general, the DNS also stores other types of information, such as a list of mail servers that accept email for a given Internet domain. This role of the DNS puts it in a sensitive spot. The user must trust the DNS server to return the correct result for

Danny Dolev is Incumbent of the Berthold Badler Chair in Computer Science. This research was supported in part by the Israeli Science Foundation (ISF) Grant number 1685/07.
This paper will appear in the Proceedings of Secure Comm 2011.

his request. If the DNS server sends an incorrect IP address to the user, the user will access a different site while assuming he is accessing the site he intended to access. This problem becomes more severe with the DNS caching system that is used by the DNS servers for speeding up the requests' processing. Attackers search for opportunities to place faulty records into the DNS's cache. Once the attacker manages to implant such a record (that is to poison the cache), every user that requests this (poisoned) record will receive an IP address of a malicious site.

The DNS protocol usually uses User Datagram Protocol (UDP) as a forth level protocol for its data communication. If for some reason the request or the response fails to reach its destination, the DNS Server simply issues another request. For such a case, the DNS Server needs to be able to handle the situation that arises from packet delays, as these may be accidently interpreted as packet losses. The DNS operates in a straightforward approach. It simply accepts and caches the first valid response (that is, a response from an authoritative server) and ignores all other responses. This is a drawback in the DNS security and a gateway for attackers to poison the cache. (See [3].)

Pharming occurs when an attacker redirects a web site's traffic to a bogus web site. Pharming is the primary risk associated with cache poisoning. Attackers employ pharming for four primary reasons [4]: identity theft, distribution of malware, dissemination of false information, and man-in-the-middle attacks.

This paper presents a *Delay Fast Packets* (DFP) algorithm which detects and prevents attempts of cache poisoning attacks. In order to successfully poison the cache, the attacker response must beat the real response (from an authoritative server) in the race back to the DNS resolver, which is the local DNS server that originated the request. In our model, we assume an eavesdropping attacker. The attacker can generate a response that is identical to the real response. Since the window of opportunity is short, the attacker tries to send a response as soon as possible and usually does so much faster than it takes the authoritative server to generate a response. Our DFP algorithm identifies that exact point by analyzing the distribution of the round trip time (RTT) from the moment the request leaves the resolver to the time the resolver gets the response. This distribution is saved for each potential authoritative server. When the algorithm identifies an anomaly in the RTT of a response, it delays the response for a short interval and waits for another response of the same request to arrive. If no additional response arrives in that interval, the delayed response is sent to the resolver.

Our contributions are two-fold. Firstly, we prevent attacks under a very strict model against a powerful adversary. To our knowledge, we are the first to introduce an engine that does not change the DNS protocol and which still assumes an eavesdropping attacker that has all the information it needs in order to generate a valid response. DNS requests and responses today are completely unencrypted and are broadcast to any attacker who cares to look. Anybody with access to the copper infrastructure can *eavesdrop*. Moreover, most of this wiring is relatively unprotected and easy to access. In fact, this strict model has a significant impact on the motivation behind solutions that encrypt the DNS packets (e.g. [5]). Existing solutions that do not change the DNS protocol do not defend a DNS server in such model (as detailed in Section 3). In addition to the strict model, our solution can be implemented as a black box that gets each request

right after it leaves the resolver. Therefore, no modifications are required, neither to the DNS protocol nor to the BIND (Berkeley Internet Name Domain) server code.

The rest of this paper is organized as follows. Section 2 describes the cache poisoning attack and the common approach to prevent it. Section 3 presents the state of the art algorithms against a cache poisoning attack. Section 4 presents our algorithm. Section 5 details the considerations we examined when we chose the algorithm parameters. Section 6 presents our experimental results, and Section 7 concludes this paper.

2 Cache Poisoning

When a client waits for a DNS response, it will only accept the information returned if it includes the client's correct source port and address in addition to the correct DNS transaction ID. These three pieces of information are the only form of authentication used to accept DNS responses. Knowing the source IP is straightforward as we know the address of the name server to be queried. The source port, however, and the transaction ID present a challenge. BIND often reuses the same source port for queries on behalf of the same name server, therefore discovering the source port is not a hard task [6]. The only real obstacle that stands between the attacker and a successful cache poisoning is the transaction ID field in the DNS protocol. Therefore, the attackers look for weak spots in the protocol implementation that can allow them to make a good *guess* of the transaction ID and, in this way, interfere with the traffic. In this section we present the methods used by the attacker to overcome this obstacle.

BIND (Berkeley Internet Name Domain) [7] is the most commonly used Domain Name System (DNS) server on the Internet. The earliest BIND servers did very little to address security. In order to avoid a same transaction ID repeating at the same time in the network, the server used an "Increment by One" method. Each new query was issued with the previous $transactionID + 1$. Guessing the transaction ID in such a case is a fairly easy job. This weakness was patched and the new BIND versions issue a random transaction ID to every new query. In the new version (BIND 9), the transaction ID is a randomly generated number, or more precisely, the transaction ID is a pseudo random generated number. The algorithm that generates the IDs in each of the BIND versions is open to the public and can be easily obtained and studied. As shown in [8], in many of the BIND 9 versions, the algorithm is weak and the next random number can be derived from the previous one. This particular problem was fixed in the 9.5.0 BIND version. Here, in order to guess the correct transaction ID, an attacker can use the *birthday paradox*. The attacker first simultaneously sends a large quantity of packets to the DNS server requesting the same Domain Name. The DNS server generates the same number of queries and sends them to the authority server. The attacker generates the same amount of DNS bogus responses with a random transaction ID. The birthday paradox dictates that a few hundred packets will suffice to promise a 50% success rate where there will be a match of the transaction ID with at least one query and one bogus response. This leads to a successful poisoning of the cache to the DNS server. Such an attack was fully described in [9]. The birthday attack guaranties high chances of success with a relatively low number of packets required. In regular packet spoofing, if the attacker sends N responses for one query, the probability of success is $\frac{N}{T}$ where T

is the total number of packets possible (in the DNS case $T = 2^{16} - 1 = 65535$). In the birthday paradox attack, the attacker only needs to match one of the requests to one of the responses. The probability of success can be calculated by the following formula:

$$P(success) = 1 - 1(1 - \frac{1}{T})(1 - \frac{2}{T})...(1 - \frac{N-1}{T}) = 1 - \frac{T!}{T^N(T-N)!} \ .$$

The power of the birthday paradox attack over the regular packet spoofing attack is that it requires a relatively small number of packets in order to make a successful attack. A mere 300 packets guarantees 50% success, while 750 packets guarantees a 99% success rate. In the regular packet spoofing attack, 750 packets only guarantees a $\frac{750}{65535} = 1.14\%$ success rate. The birthday paradox attack shows that even a randomly generated transaction ID used in the latest BIND versions is vulnerable to brute-force attacks.

The big security news of Summer 2008 has been Dan Kaminsky's discovery of a serious vulnerability in DNS servers [10]. In this exploit, the attacker causes the target name server to query for random host names at the target domain. The attacker can spoof a response to the target server including an answer for the query, an authority server record, and an additional record for that server, causing the target name server to insert the additional record into the cache.

There are several solutions available for the problem of a cache poisoning attack as presented in Section 3. In our algorithm we assume the attacker knows the transaction ID, source port, or any other information from the request needed in order to generate a valid response. In contrary to other solutions, we are not trying to increase entropy, rather we assume it is known to the attacker. The presented algorithm detects anomalies in the RTT of the responses. Since in order to get into the cache, a spoofed response has to arrive before the correct one, the RTT of those responses is shorter than it usually is and therefore is considered anomalous.

3 Related Work

There are several available solutions on how to prevent cache poisoning attacks and attempts. In this section we present some of them. BIND is the most widely used DNS software over the Internet [1], [2], and therefore it is a constant target to attackers' attacks. New versions and version updates are constantly being released constantly with new updates and patches for bugs and security issues. Therefore the easiest way to enhance the security of a local DNS server is to run the most recent version of BIND.

DNS security solutions can be categorized into two categories. The solutions in the first category extend the existing DNS protocol. Solutions in the second category require massive changes and thus new DNS servers deployment. Since a large-scale deployment may not be reached in the near future, an extensive search is made in order to design solutions that do not require new deployment.

A lot of effort has been spent in trying to make the DNS transaction ID more random and less predictable [11], [12]. Ultimately, such efforts are insufficient since with only 16 bits to fight over, a determined attacker can use a purely random attack, or even a constant attack, and theoretically, eventually, and statistically speaking, break

through the requestor's defenses. Most of the research these days is based on increasing the entropy of DNS queries in order to make forging a valid response more difficult. In [13] [14], the authors describe a method by which an initiator can improve transaction identity using the 0*x*20 bit in DNS labels. This idea uses the question section to add random bits to the query. DNS servers do not care if the question is presented in upper or lower case, and therefore a combination of the cases can provide the essential random bits to the query. In practice, all question sections in responses are exact copies of question sections from requests. The difference between lower and upper case letter is the 0x20 bit. Therefore, for any character in the domain name in the question, a request initiator can randomly choose this bit and the transaction ID can be effectively lengthened beyond 16 bits. The effectiveness of this algorithm is a function of the length of the domain. In the Random prefix [15], [16] method, the authors propose to use wildcard Domain Names to increase the entropy. For example, if a user wants to resolve the "www.example.com", the DNS server will generate a random prefix for the query and send "ra1bc3twqj.www.example.com". The authoritative DNS server returns the same domain name with the "www.example.com" IP address. This method using a prefix length of 10 will generate in the region of $\log_2 36^{10} \approx 52$ bits. In another solution, presented in [17], the authors extend the DNS query ID with up to 63 alpha-numeric characters into the query/response question name (QNAME) making the range of possible transactions IDs so large that any brute force guessing or birthday attack attempts are futile.

Most name-servers, prior to the patches released on July 2008, always sent out their queries from port 53. Therefore, another direction is to also randomize the source port [18], [19], [12]. In this method, the name server uses a random source port for his query. The name server cannot use an entire UDP port space, however, even an extra 10 or 11 bits of randomness is many times greater. A DNS source port randomization becomes vulnerable if the DNS traffic is behind NAT. NAT cancels the DNS source port randomization by translating source ports to non-random ports.

Since the DNS protocol does not include any security, Domain Name System Security Extensions (DNSSEC) [20] were developed as described in RFC 3833 [21]. DNSSEC was designed to prevent cache poisoning by having all its answers digitally signed, thereby allowing the correctness and the completeness of the data to be easily verified. DNSSEC is a new protocol and only lately have some of its critical pieces been formally defined. Using DNSSEC necessarily means deploying new servers or reinstalling the protocol in the existing ones. Consequently, deploying the protocol on large-scale networks becomes a challenging task. DNSSEC introduces new security issues such as chain of trust problems, timing and synchronization attacks, Denial of Service amplification, increased computational load, and a range of key management issues as presented in [22].

DNSCurve [5] is an alternative to DNSSEC. DNSCurve uses high-speed elliptic curve cryptography, and simplifies the key management problem that affects DNSSEC. There is not much documentation on DNSCurve, but like DNSSec, it is hard to deploy.

4 The DFP Algorithm

The primary aim of the DFP algorithm is to estimate the RTT (Round Trip Time) between the DNS Server and each of the authoritative servers it encounters and to delay the responses that are arriving *too fast* according to the approximation. Furthermore, the processing time for each service type (MX, A, AAAA, CNAME, PTR etc...) might have different lengths, such as in a case due to a more extensive database search on the authoritative side. Therefore, the DFP algorithm estimates the RTT for each service type the authoritative server can provide. For each authoritative server and service type, the estimated RTT predicts the average time needed for the next response to arrive. If for any reason, a response comes too soon, according to the DFP algorithm, the DNS Server waits for a certain amount of time before it forwards the response to the requestor. If another valid response arrives in that window of time, both responses are dropped, and a new request is generated (as is done when a regular DNS packet loss occurred). If the attacker is persistent and sends a response for each request, the user experiences DoS (Denial of Service) attack, since the DFP algorithm will not pass any of the responses back to the user. In this case, the user does not get the service, but at least he is also not exposed to more harmful attacks such as fishing and theft of critical information. Moreover, under the assumption of an eavesdropping attacker and without changing the DNS protocol, we believe that there is no solution that can also solve the DoS problem. A simple cache poisoning attack with an eavesdropping attacker is presented in Figure 1. A local name server that is deployed with the DFP engine is not vulnerable to a cache poisoning attack as shown in Figure 2.

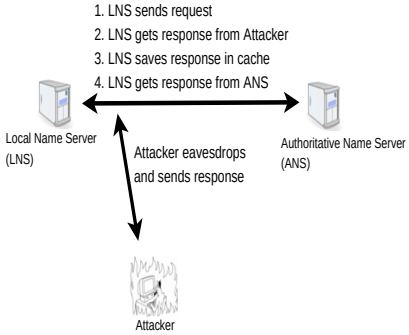


Fig. 1. Cache Poisoning Example

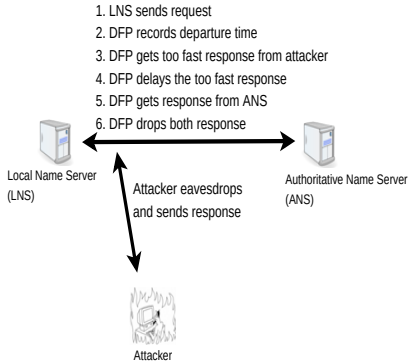


Fig. 2. DFP Operation

Algorithm 1 presents a simplified pseudo-code that demonstrates the idea of the DFP algorithm. In the case of a multiple packet attack, the algorithm *closes* the request after the first duplicate response, so any other response will not have a corresponding request and, thus, will be dropped. Another issue we have to consider is the legal *too fast* packets that might affect the RTT estimations. In the case where there are no attacks, those *too fast* packets can mark a change in the topology of the network and therefore

Algorithm 1 DFP - Delay Fast Packets

```
1: PacketDictionary.Init() //mapping responses to request
2: StatsDictionary.Init() //save auth. server statistics
3: loop
4:    NewPacket ⇐ SniffDNSPacket()
5:    key ⇐ GetKey(NewPacket.Auth−server, NewPacket.TransactionID, NewPacket.Type)
6:    if NewPacket.isQuery() then
7:       PacketDictionaly.put(key, NewPacket)
8:    else
9:       RequestPacket ⇐ PacketDictionary.get(key)
10:      if RequestPacket == NULL then
11:         Drop(NewPacket)
12:      else
13:         if RequestPacket.hasDelayedResponse() then
14:            Drop(DelayedResponse)
15:            PacketDictionary.clear(key)
16:            Drop(NewPacket)
17:         else
18:            RTT ⇐ NewPacket.TimeOfArrival − RequestPacket.TimeOfSend
19:            DelayTime ⇐ AuthServerStats.AddSample(RTT, NewPacket.Auth −
               Server, NewPacket.Type)
20:            DelayPacket(DelayTime)
21:            PacketDictionary.clear(key)
22:         end if
23:      end if
24:   end if
25: end loop
```

AddSample(RTT, Auth-Server, Type)

```
1: AuthServerStats = StatsDictionary.get(Auth−Server+Type)
2: if AuthServerStats == NULL then
3:    AuthServerStats = CreateStat(key)
4:    AuthServerStats.EstimatedRTT ⇐ RTT
5:    AuthServerStats.DevRTT ⇐ 0
6: end if
7: AuthServerStats.EstimatedRTT ⇐ (1−α)×AuthServerStats.EstimatedRTT+α×RTT
8: AuthServerStats.DevRTT ⇐ (1 − β) × AuthServerStats.DevRTT + β × |RTT −
   AuthServerStats.EstimatedRTT|
9: if RTT < AuthServerStats.EstimatedRTT − AuthServerStats.DevRTT ×
   AuthServerStats.FactorWindow then
10:   return (AuthServerStats.EstimatedRTT + AuthServerStats.DevRTT ×
      AuthServerStats.FactorWindow)−RTT
11: else
12:   return 0
13: end if
```

must be considered in the RTT estimations. However, in the case of possible attacks, the algorithm should not include them in the estimations, as they may be an attempt of the attacker to lower our RTT estimations in order to make a successful attack in the near future. Therefore, *too fast* packets must not affect the RTT until the algorithm verifies their authenticity. This functionality is omitted from the the pseudo-code in order to save its simplicity.

The algorithm was tested on real traffic from the local DNS server of our university. The traffic contains $385,000$ DNS requests.

The DFP algorithm uses two hash tables. *PacketDictionary* maps between the outgoing requests and the incoming responses; *StatsDictionary* stores the statistics for each authoritative DNS server. On each packet arrival, a key is constructed from the authoritative server IP, the transaction ID, and the packet type. If the packet is a request, the packet is saved, by its key, in the PACKETDICTIONARY hash table. If the packet is a response, the corresponding DNS request is retrieved, once again, by the same key. The scenario when no matching request is found, that is, there is a response with no request, can be a result of two cases. One, the attacker sends multiple responses and the request was previously cleared. Two, there is a response without a request. In both cases, this condition can never be fulfilled unless there is an attack on (or a bug in) the DNS server; therefore, the packet is dropped. If the corresponding request has a delayed response (a *too fast* response was previously arrived to that request), the algorithm removes the request from the PACKETDICTIONARY hash table and drops both responses. In the normal case, where both the response and the request are found, the algorithm calculates the RTT between the local DNS server and the authoritative DNS server by measuring the time difference between the time the request is sent to the arrival of the response. Note that the RTT is calculated for each authoritative server and service type. It then calculates the ESTIMATEDRTT DEVRTT and estimates the normal window. If, however, the packet is *too fast*, it is delayed for d milliseconds, where d is the deviation between the RTT and the upper bound of the estimated normal window. Otherwise, the response is immediately sent to the server to be saved in the cache.

5 Design Parameters

The DFP algorithm uses the following formula in order to detect *too fast* packets: $RTT < EstimatedRTT - DevRTT \times FactorWindow$. Each DNS response that arrives too soon according to the formula is considered suspicious and delayed, thereby allowing time for another possible response with the same transaction id to arrive. The variables in the formula are controlled by three parameters: α, β and $FactorWindow$. The performance of the DFP algorithm, in terms of speed, detection accuracy, and memory consumption, depends on how well these parameters are configured. In this section we describe the considerations and the experiments that led us to choose the values for these three parameters.

5.1 The Window Parameters

The **Window** is the time interval in which response arrivals are considered normal. Each response that arrives before the window begins is considered suspicious. Each

pair of authoritative server and request type has its own window. For example, for www.abc.com authoritative DNS server with type A, the window might begin 3400 ms after the request is sent, while for type MX it might begin after 3800 ms. The **Window Starting Point** is the beginning of the window. Each response arriving before the starting point is considered as a *too fast* packet. Respectively, the **Window Ending Point** is the end of the window and each response arriving after the ending point will be considered as a *too slow* packet. A false alarm occurs when a packet originated by the authoritative server arrives before the window starting point. Every false alarm causes the DNS server to store the packet in memory for a short time and release it only after it is safe. Figure 3 presents these parameters over the time axis. The sending time of a specific request is $t_{send\_request}$. The average of the arrivals times of all legal responses for the specific request type and authoritative name server is $t_{avg\_responses}$. This average has margins that define the window's starting point and ending point. Any response that arrives between $t_{send\_request}$ to W_{start_point} is considered *too fast* and any response that arrives after W_{end_point} is considered *too slow*. Some authoritative servers are infrequently requested and due to the dynamics of the network the DFP algorithm might not have enough samples in a certain point to create a distribution. The algorithm either takes the minimum values of the window starting point and window ending point, if they exist, or it takes the minimum values of an authoritative name server from the same parent domain.

The window has a very dynamic nature. Its starting point constantly changes and shifts on the time axis. This is due to the dynamic nature of the internet network and the constant changes in the RTT of the arriving requests. The window starting point dictates which packets are considered *too fast*, and which thus need to be delayed, and which packets are within the normal time boundary and can therefore immediately pass through. An attacker might try to influence the location of the window by flooding the authoritative server. In this case, the latency of the responses from the flooded authoritative server increases and the window is shifted to the right, resulting in a delayed starting point and fewer chances to successfully poison the DNS cache. In order to adjust the parameters that define the window starting point, there are two observations to consider:

- An early starting point allows more packets to pass through without a delay. The DNS server does not need to delay too many suspected packets (until it is safe to pass them on) and therefore the latency is reduced. However, the window of opportunities is increased, and a potential attacker can hit just above the starting point and pass the filter without triggering an alarm.
- A late starting point delays more packets since it considers them as *too fast* packets. This configuration hardens the attacker cache poisoning attempt since in order to avoid the DFP filter he has to compete on a small time interval. However, a late starting point forces the DNS server to delay many packets, considering them as potential threats. The major consideration of this configuration is the larger memory consumption and a slower response of the DNS server to the users.

In the following sections (5.2 and 5.3) we refer to α, β and the *FactorWindow*. We perform a set of experiments in order to demonstrate the influence of each of the

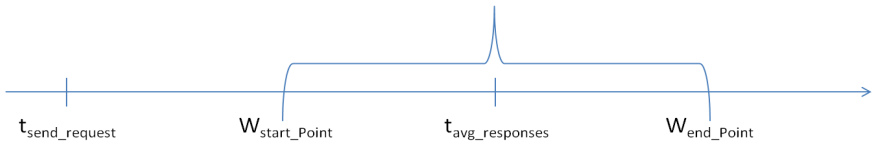


Fig. 3. The window Parameters

parameters on the window starting point and hence on the tradeoff between the number of false alarms and the probability of detecting and preventing a potential attack.

5.2 α and β Considerations

The two parameters influencing the EstimatedRTT and the DevRTT parameters in the DFP algorithm are α and β. They determine the weight of the new RTT sample against the history, thereby influencing the window starting point. In order to find how α and β influence the number of fast packets detected by the DFP algorithm, we conducted several experiments on real traffic without any attempted attacks. In each experiment we measured the number of false positives alarms. The following figures 4, 5, 6 demonstrate the results of the experiments, using different values of α and β. In order to clearly demonstrate the results, the graphs present only 100 packets that represent the general case.
Note: The *FactorWindow* parameter is set to 2 in each of the following experiments.

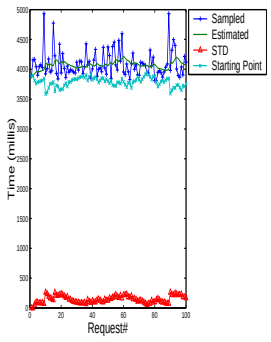


Fig. 4. α=0.125, β=0.25

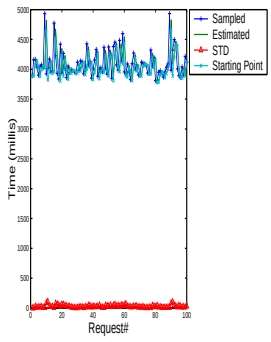


Fig. 5. α=0.875, β=0.75

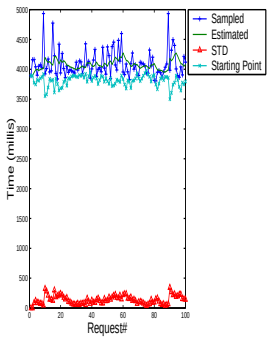


Fig. 6. α=0.2, β=0.4

In each of the experiments, the newest sample is given a much higher weight since it is better at predicting the future RTT. Figure 4, deals with the case of low values of α and β. Low α and β values, as in TCP RTT estimation, smooth the estimated RTT function, since more weight is given to the history of the samples rather than to the newest sample (in comparison to higher values of α and β). For each peak in

the sampled RTT, the DevRTT rises. (Note, for example, the peak in the sample RTT and the rise of DevRTT at packet number 10.) As a result, the window starting point becomes low. This situation allows potentially malicious responses a wider window of opportunity to attack the DNS server. Only two packets were considered *too fast* in this configuration. The graph shows that those packets' RTT time exceeded the starting point. In Figure 5, we used high values of α and β. High values give most of the weight to the newest sample. Hence, the RTT deviation is very small and the window starting point is extremely late, making DNS attacks attempts very hard to succeed. However, this situation also creates many false alarms as any fluctuation in the RTT will probably put the new sample before the window starting point. We see that in this configuration about 20% of the packets were considered *too fast*. Figure 6 deals with the case of medium values of α and β. The values of α and β are the median of the 'Low' and 'High' configurations. As expected, the window starting point in this case is later than in the 'Low' configuration and the RTT deviation is higher than in the 'High' configuration. We see that in this configuration five of the packets were considered *too fast*.

Our experiments show that most of the time the deviation of the RTT is relatively low. Therefore the created starting point is rather high. The change in the deviation occurs when an extremely slow packet arrives. In this situation, the window starting point is lowered for a short period of time and possible attacks have a higher chance of success. However, as we can see from the results, slow packets seldom arrive. In consideration of memory consumption, it is important to prevent false alarms that might be created by valid *too fast* packets. Thus, for those, the 'Low' version should be chosen. However, if the local DNS server can afford saving more *too fast* packets, the better configuration is the one that prevents more attacks, and in that case it is better to choose the 'Medium' or even (if memory is not a problem) the 'High' configuration.

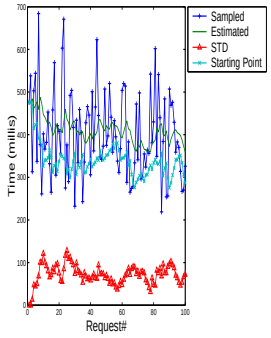
5.3 FactorWindow Considerations

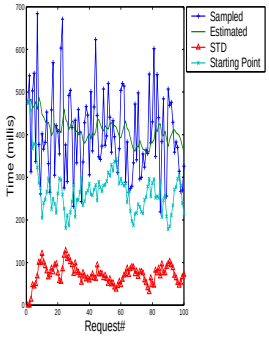
After setting up the α and β parameters, the configuration of the *FactorWindow* parameter should be determined. This parameter goal is to lower the starting point created by α and β. As before, the tradeoff between the number of false alarms and the probability of a successful attack dictates which value will be chosen. Figures 7, 8 and 9 present the *FactorWindow* influence on the window starting point. As above, in order to clearly demonstrate the results, the graphs present only 100 packets that represent the general case.

Note: The α and β parameters are set to 0.125 and 0.25 respectively in each of the following experiments.

Figure 7 deals with the case where $FactorWindow = 1$. $FactorWindow = 1$ means that the window starting point is modified only by α and β. Therefore the created starting point is high and the probability for a packet to come before the starting point is respectively high. In this case, many packets will have to be delayed. We see that in this configuration, about 30% of the packets are considered *too fast* packets.

In Figure 8 we used $FactorWindow = 2$, i.e. the starting point is two estimated deviations from the estimated RTT. Using the 'Chebyshev inequality' the probability for a packet to exceed the starting point is less than $\frac{1}{4}$. However, in practice, the bound





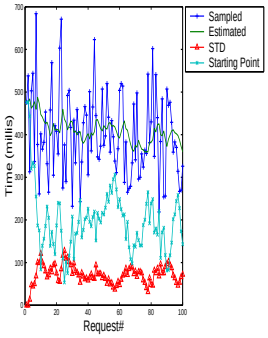


Fig. 7. Factor Window = 1 **Fig. 8.** Factor Window = 2 **Fig. 9.** Factor Window = 3

is tighter. Our experiments show that only about $\frac{1}{10}$ of the packets are considered as *too fast*.

Figure 9 deals with the case where $FactorWindow = 3$. Again, using the 'Chebyshev inequality' the probability for a packet to exceed the starting point is less than $\frac{1}{9}$, but in practice, almost no packet exceeds the starting point. The window starting point is so low that an attacker can easily intrude even without knowing that a detection and prevention DFP algorithm is running. In this configuration, only one packet is considered *too fast*.

The main consideration for choosing the configuration of the $FactorWindow$ parameter is, again, the tradeoff between the number of false positives and the probability of a successful attack. By analyzing our results, we conclude that the best value for the $FactorWindow2$ is 2.

5.4 Slow Packets Consideration

The main assumption of the DFP algorithm is that the deviation from the EstimatedRTT approximates zero. This assumption was proven to be true in many experiments carried out on real traffic. But in some cases, the deviation rises for short periods of time. In those moments, the attacker gets an opportunity for a successful attack since many *too fast* packets fall after the window starting point of : $EstimatedRTT - DevRTT \times FactorWindow$ bound and are therefore considered normal.

This situation occurs after a very slow packet is received. The *too slow* packet creates a temporary increment of the deviation and lowers the starting point, as seen in Figures 10 and 11. The starting point returns to normal parameters after a few packets, when the influence of the *too slow* packet weakens. The temporary lowering of the starting point creates an opportunity for a attacker to attack the DNS server.

The way to prevent this weakness is to eliminate the *too slow* packets from the calculation of the deviation, thereby preventing the temporary lowering of the starting point. However, the DFP algorithm must take into consideration the possibility of rapid changes in the network characteristics or topology. Thus, DFP distinguishes between seldom *too slow* packets to a real tendency and the *too slow* packets are considered

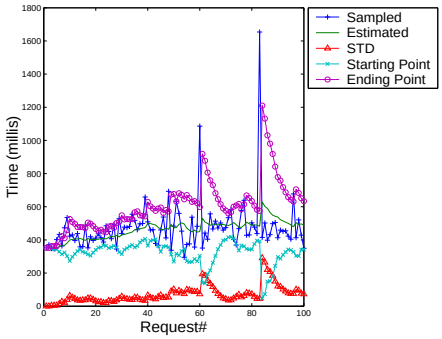


Fig. 10. Low Starting Point

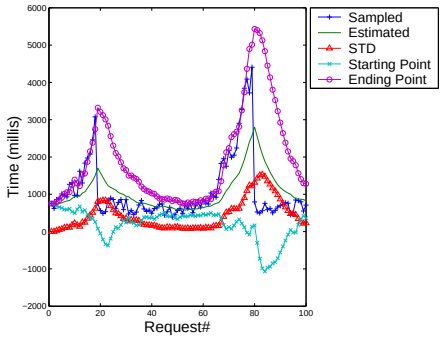


Fig. 11. Negative Starting Point

accordingly. An attacker cannot reduce the algorithm starting point by sending *slow* responses. The original response is likely to arrive before any spoofed slow response and therefore either the spoofed response is just dropped (if it arrives after the window ending point) or both the original and spoofed responses are dropped (if the spoofed response arrives within the window). In either case, the spoofed response is not considered when calculating the distribution parameters. Another option the attacker has is to flood a specific authoritative server in order to force *slow* responses from that server. As a result, the window starting point in the local name server is reduced and the attacker can send a spoofed response without being delayed. The DFP engine does not handle these kinds of combined attacks.

5.5 Imitation of the DFP Profile

An eavesdropping attacker may adopt the DFP algorithm and imitate the same profiles. Afterwards, the attacker can apply fine control on the issuing time of forged DNS responses to make them reach the server after the starting point. In order to successfully poison the cache, the attacker's response needs to arrive before the real response. The RTTs are distributed normally, therefore, if x is the arrival time of the attacker response and $t_{get\_response}$ is the arrival time of the real response, the probability of a successful attack (after standardizing x) is

$$\int_{W_{start\_point}}^{t_{get\_response}} \frac{1}{\sqrt{2\pi}} e^{\frac{1}{2}x^2}\,\mathrm{d}x\,.$$

We can see, there are two factors that influence the odds of a successful attack, the window starting point and the arrival time of the real response. We have no control over the arrival time of the real response, but we can decrease the *FactorWindow* to narrow the window of opportunity of the attacker. This scenario demonstrates the tradeoff between memory and accuracy.

6 Experimental Results

The results are measured by two factors, memory consumption and accuracy. Usually, there is a tradeoff between these two factors. In our case this tradeoff is insignificant.

6.1 Memory Consumptions

The main consideration in choosing the best configuration for the DFP algorithms is to prevent attacks. In order to prevent attacks, the starting point should be as tight as possible to the estimated RTT. However a tight starting point might create many false alarms (as explained above). In this section we present how different starting point values affect the memory consumption. We examined three configurations. In the first one, $\alpha = 0.125$, $\beta = 0.25$; in the second, $\alpha = 0.2$, $\beta = 0.4$; and in the third configuration, $\alpha = 0.875$, $\beta = 0.75$. Figure 12 presents the percentage of packets that are considered *too fast* in each of the configurations.

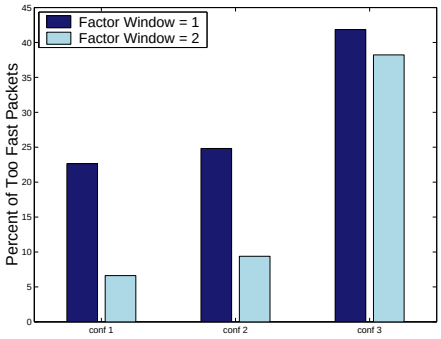


Fig. 12. Percentage of Fast Packets

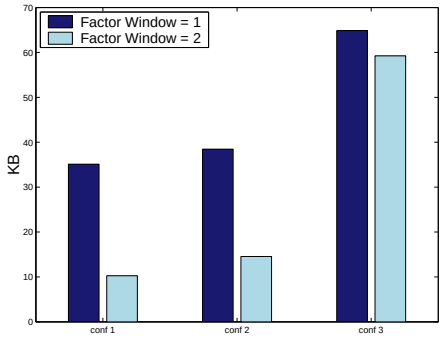


Fig. 13. Memory Consumption

The DNS payload has a limit of 512 bytes (for IPv4). Our experiments show that an average response is about 155 bytes long. The DFP algorithm must allocate those bytes in memory for each delayed packet, usually for about few hundred ms, until the response is either released or dropped. The memory consumption depends on the inbound rate of the local DNS server. The university DNS server can only handle a few dozen responses in parallel. Since this server might not represent the general case, Figure 13 estimates how many KB the DFP algorithm consumes assuming it handles 1000 responses in parallel. As we can see, even for the most wasteful configuration, the memory consumption is no more than 65KB on average and 215KB in the worst case. Thus, memory consumption is not a limiting factor even for busier servers.

The presented algorithm was implemented and tested on real traffic collected from our university DNS Server. The traffic was sniffed and saved in pcap files that were later used for different configurations testing and analysis. The traffic was filtered to contain only DNS responses with an authoritative flag on. For each of the samples, the

algorithm calculates the EstimatedRTT, the DevRTT, and with a given FactorWindow, it deduces which packets are considered to be *too fast*.

6.2 Attacks Detection

In order to test the DFP algorithm we planted a few random duplicate response packets with random arrival times in the tested traffic. The spoofed responses arrived before the real responses. The DFP algorithm with the above configuration was able to classify all of the attacks as *too fast* packets and therefore delayed them until the real result arrived. We believe that there were no real attempts to attack our university local DNS server while the samples were captured, since no duplicate packets were found beside the faked packets planted by us. Unfortunately, we cannot compare our results to other solutions since all other solutions fail to protect the DNS server from cache poisoning attack on our strict model.

7 Conclusions

This paper presents the DFP algorithm against DNS cache poisoning attacks. The algorithm assumes an eavesdropping attacker that can see the request and therefore can easily create and send a spoofed response. Our algorithm measures statistics per authoritative server and type of query in order to build a profile about the RTT distribution for these two parameters. Since, in order to get into the cache, a spoofed response has to arrive before the correct one, the RTT of those responses is shorter than it usually is and therefore, out of the constructed profile. We showed that the algorithm is scalable and its memory consumption can fit in a standard cache.

The weak spot of the DFP engine is its vulnerability to a DoS attack (in the case where the attacker repeatedly sends spoofed responses). In our future work, we will integrate the DFP engine with a mechanism that detects these repetitive spoofed responses and instead of just dropping duplicate responses, it will save a copy of each unique response and choose the correct one according to various considerations.

References

1. “Dns,” http://www.ietf.org/rfc/rfc1034.txt.
2. “Dns,” http://www.ietf.org/rfc/rfc1035.txt.
3. C. Schuba, “Addressing weaknesses in the domain name system protocol,” Master’s thesis, august 1993.
4. R. Hyatt, “Keeping dns trustworthy,” The ISSA Journal, pp. 37–38, 2006.
5. D. J. Bernstein, “Dnscurve,” http://dnscurve.org/.
6. Sainstitute, “Attacking the dns protocol security paper,” 2003.
7. D. B. Terry, M. Painter, D. W. Riggle, and S. Zhou, “The berkeley internet name domain server,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-84-182, May 1984. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5957.html
8. A. Klein, “Bind 9 dns cache poisoning,” 2007.

9. J. Stewart, “Dns cache poisoning - the next generation,” 2003.
10. D. Kaminsky, “The kamisky bug,” http://dankaminsky.com/.
11. I. Internet Systems Consortium, “Bind 9,” http://www.bind9.net/, 2003.
12. “Powerdns,” http://doc.powerdns.com/, 2011.
13. P. Vixie and D. Dagon, “Use of bit 0x20 in dns labels to improve transaction identity,” http://tools.ietf.org/html/draft-vixie-dnsext-dns0x20-00, 2008.
14. D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee, “Increased dns forgery resistance through 0x20-bit encoding: security via leet queries,” in Proceedings of the 15th ACM conference on Computer and communications security, ser. CCS ’08. New York, NY, USA: ACM, 2008, pp. 211–222. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455798
15. R. Perdisci, M. Antonakakis, and W. Lee, “Solving the dns cache poisoning problem without changing the protocol,” 2008.
16. R. Perdisci, M. Antonakakis, X. Luo, and W. Lee, “Wsec dns: Protecting recursive dns resolvers from poisoning attacks.” in DSN. IEEE, 2009, pp. 3–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/dsn/dsn2009.html#PerdisciALL09
17. J. G. Hoy, “Measures for making dns more resilient against forged answers,” http://www.jhsoft.com/dns-xqid.htm, 2008.
18. A. Hubert and R. van Mook, “Anti dns spoofing - extended query id (xqid),” http://tools.ietf.org/html/draft-ietf-dnsext-forgery-resilience-10, 2008.
19. “djbdns,” http://cr.yp.to/djbdns.html, 2004.
20. “Dnssec,” http://www.dnssec.net/.
21. D. Atkins and R. Austein, “Threat analysis of the domain name system (dns),” http://www.ietf.org/rfc/rfc3833.txt, 2004.
22. S. Ariyapperuma and C. J. Mitchell, “Security vulnerabilities in dns and dnssec,” in Proceedings of the The Second International Conference on Availability, Reliability and Security. Washington, DC, USA: IEEE Computer Society, 2007, pp. 335–342. [Online]. Available: http://portal.acm.org/citation.cfm?id=1249254.1250514