Vulnerable Delegation of DNS Resolution

Amir Herzberg[1] and Haya Shulman[2,⋆]

[1] Computer Science Department,
Bar Ilan University, Ramat Gan, Israel
[2] Fachbereich Informatik,
Technische Universität Darmstadt/EC-SPRIDE, Darmstadt, Germany
{amir.herzberg,haya.shulman}@gmail.com

Abstract. A growing number of networks delegate their DNS resolution to trusted upstream resolvers. The communication to and from the upstream resolver is invisible to off-path attackers. Hence, such delegation is considered to *improve* the resilience of the resolvers to cache-poisoning and DoS attacks, and also to provide other security, performance, reliability and management advantages.

We show that, merely relying on an upstream resolver for security may in fact result in vulnerability to DNS poisoning and DoS attacks. The attack proceeds in modular steps: detecting delegation of DNS resolution, discovering the IP address of the internal (proxy) resolver, discovering the source port used for the (victim) DNS request and then completing the attack. The steps of the attack can be of independent use, e.g., proxy resolver can be exposed to denial of service attacks once its IP address is discovered.

We provide recommendations for securing the DNS service delegation, to avoid these vulnerabilities.

Keywords: network security, DNS cache poisoning, port randomization.

1 Introduction

Increasingly, organisations delegate sensitive network-operation functions to trusted providers ('clouds'). The motivations are diverse, and include cost-savings, efficiency, reliability, and even security, i.e., the trusted (cloud) provider is deemed to be able to provide good or even better security. In this work, we study a particularly important type of such delegation: the use of *upstream DNS resolver*, e.g., OpenDNS[1]. An *upstream resolver* is a DNS resolver operated by a trusted provider, outside of the customer's network, and used directly by the customer's *DNS proxy* (local resolver) [2].

⋆ This work was carried out while the second author was in the Department of Computer Science, Bar Ilan University.

[1] Many upstream resolvers are *open/public*, i.e., provide resolution service to any Internet client address, e.g., OpenDNS and Google-public-DNS.

[2] Some authors use the term 'forwarder' for upstream resolver, while others use this term for DNS proxy (local) resolver. To avoid confusion, we use the 'upstream' and 'proxy' resolver terms throughout this work.

Since DNS services are critical and often attacked, upstream DNS resolvers are increasingly adopted by many networks, and recommended by leading experts and vendors, e.g., Google, OpenDNS, Comodo and Akamai. For example, Akamai describe their upstream DNS service, dubbed *eDNS*, as follows [1]:

'Using eDNS, a customer's primary DNS servers are not directly exposed to end users, so the risks of cache-poisoning and denial-of-service attacks are mitigated.'

In this work, we show that typical use of upstream DNS resolvers, can actually result in *illusion of security* exposing to *DNS cache poisoning and DoS vulnerabilities.*

To understand the potential *loss* of security due to the use of upstream DNS resolvers, we next briefly discuss a simple case: *DNSSEC validation.* DNSSEC, [RFC4033-4035], is a standard for signing DNS records, allowing resolvers to validate DNS responses, and hence ensuring security against man-in-the-middle (MitM) attackers. So far, DNSSEC is not widely deployed, both at the zones as well as at the resolvers. For example, Google reports that less than 1% of the DNS records it retrieves are signed; and [2] tested queries to `org` and found that 0.8% of the resolvers were validating. Clearly, the deployment of DNSSEC is still very limited; we hope that our results will encourage wider adoption.

Some upstream DNS resolvers, e.g., Google's public DNS, perform DNSSEC validation. How does that effect the security of their customers? Clearly, since these upstream resolvers validate responses, this prevents attacks where false responses are sent to the upstream resolver. However, the proxy resolver may be vulnerable to an attacker sending forged responses directly to the proxy resolver, unless the the proxy resolver will also perform validation. In this way, the use of upstream resolver may cause reduction of security, due to illusion of security (by the upstream resolver).

The use of upstream resolvers was recommended by Kaminsky and other experts, e.g., [3], as a defense against his off-path attack [4]. Indeed this configuration is believed to defend against cache-poisoning attacks [5], and as a result many proxies (that use upstream resolvers) are not patched. Furthermore, DNS checker services, e.g., [6,7,8], designed to check if resolution services are secure, are oblivious to the proxy-behind-upstream resolver scenario, and do not report a problem, even when the proxy is using fixed a source port. Although many studies report that resolvers adopted port randomisation, recommended in [RFC5452], those statistics do not apply to proxies-behind-upstream resolvers, most of which use fixed or predictable ports; we confirmed this using CAIDA's data traces [9]), see Section 4.

We show that, in contrast to folklore belief, customers whose proxy resolver uses (a secure) upstream resolver for DNS services, with or without DNSSEC validation, may actually be susceptible not only to a MitM attacker, but even to an off-path attacker. Namely, such customers may fall victim to *illusion of security.*

Attacker Model. We assume an *off-path attacker* on the Internet that can send packets with a spoofed source IP address. The attacker also controls a weak, 'puppet' (sandboxed client) [10], such as a client running scripts or presenting Flash content.

Related Work. Cache poisoning poses one of the significant threats to DNS and to Internet infrastructure. DNS poisoning can facilitate many other attacks, e.g., injection of malware, phishing, website hijacking/defacing, circumventing same origin policy. The main technique for DNS poisoning (by the common *off-path* attackers) is by generating spoofed responses to DNS requests which were sent by resolvers. The best defense against DNS cache poisoning is cryptographic authentication of the responses, using DNSSEC [RFC4033-4035]. In addition to preventing attacks by off-path attackers, DNSSEC also defends against MitM attackers. Unfortunately, DNSSEC is not widely deployed and most resolvers use challenge-response mechanisms as a defense against off-path attackers, i.e., resolvers validate that the response echoes some unpredictable (random) values sent within the request, such as the DNS transaction ID (TXID) field and the source port, see [RFC5452] for more details; firewall-based defenses were also proposed against poisoning [11].

Significant research effort was dedicated to identifying vulnerabilities allowing off-path attacks, and improving defenses. We next review the main results.

Klein [12] showed that some implementations use weak TXID values which can be predicted. Indeed, as pointed out by Vixie [13] already in 1995, the TXID field alone is simply too short (16 bits) to provide sufficient defense against a determined *off-path* attacker, who can foil it by sending multiple spoofed responses. Bernstein [14] suggested to improve the defense against spoofed responses by sending DNS requests from *random ports*, which can add a significant amount of entropy. To prevent the *birthday attack*, where attacker causes resolver to issue multiple requests for the same domain in order to increase the probability of a match with one of the spoofed responses, Bernstein [14] and others suggest to limit the maximal number of concurrent requests for the same resource record.

Many implementations did not integrate support for these suggestions till the Kaminsky attack, [4], which showed that DNS cache poisoning was a practical threat, by leveraging the known birthday weakness and the fact that TXID is too short, in tandem with an innovative method allowing to repeat the attack without the cache limitation (rather than waiting for the cached record to expire). As a result, it became obvious that changes were needed to prevent DNS poisoning. Indeed, most DNS resolvers were either patched or configured to use a patched upstream resolver. The most basic patches are source port randomisation and birthday protection, [RFC5452]. Recently we showed, [15], that NAT devices that support port randomisation recommended in [RFC6056] are vulnerable to port derandomisation attacks, and expose resolvers to cache poisoning; we also presented techniques, [16,17], allowing to circumvent other patches.

In this work we show that the recommendation to (merely) rely on (a patched and secure) upstream resolver may also fail to ensure security and such proxy resolvers may yet be vulnerable to (off-path) poisoning. In particular, we show how off-path attackers may find the IP address of the proxy resolver, as a first step in a poisoning attack or for denial-of service attacks. This is in spite of the fact that upstream resolver are hidden from attackers, since their IP address is not visible; we show how off-path attacker can find the IP address of the proxy and abuse it, e.g., for denial-of service attacks.

Our contributions and observations. This work has the following contributions:
▶ We present technique that allows to detect the use of upstream resolvers (Section 2).
▶ We show how an off-path attacker can find the IP address of a proxy resolver, contradicting the belief that the use of an upstream resolver hides the address of the proxy (e.g., to defend against DoS attacks[3]); see Section 3.
▶ In Section 4, we present efficient and practical techniques that allow an off-path attacker to find the source port of a proxy resolver. Our attacks apply to proxies that are configured to use an upstream resolver, and are effective for the common case when the proxy supports either a fixed or sequentially incrementing source ports. We also show how to extend our attacks, so that they apply to resolvers connected directly (without upstream resolver), for the common case of *per-destination incrementing* source ports, recommended in [RFC6056].
▶ We conducted measurements on CAIDA data traces [9], and found that *multiple* proxies use fixed or sequential source ports, and are thus vulnerable; see Section 4. Since many name servers, that collect statistics on DNS requests, report that random ports are widely deployed, our statistics also imply that fixed (or sequentially incrementing) ports, well known to be vulnerable, are more commonly used for proxies (using upstream resolvers) than for resolvers making direct requests to name servers.
▶ The best defense against DNS poisoning is to use DNSSEC validation (on the clients), and we hope that this paper will help advance adoption of DNSSEC. However, since DNSSEC adoption is not trivial and may take a long time, and since DNSSEC does *not* prevent the DoS attack (when the proxy's IP address is exposed), we present several efficient defenses against all of the vulnerabilities in this work, in the full version of this manuscript [18].

2 Detecting an Upstream Resolver

In this section we show that it is possible to detect whether DNS resolution on a client's network is done using an upstream resolver, using a puppet (e.g., sandboxed script) running on the client. This allows an attacker to detect if the network is vulnerable to our attacks. The detection can be used for benign purposes, e.g., collecting statistics, however we focus on its use as part of an attack; in this case, the measurements are done by an attacker generating requests from a client (e.g., using puppet), to a name server operated by the attacker. The detection exploits the fact that when using an upstream resolver, the (round trip) delay $\varDelta_R$ for a complete DNS resolution, from client to attacker's name

[3] One of the advantages of using upstream resolvers is that proxies have limited bandwidth. Therefore, launching denial of service against proxies is often significantly easier.

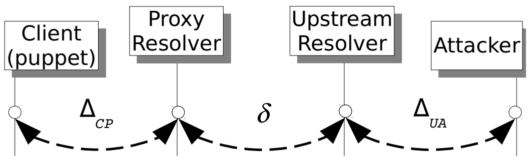


Fig. 1. The latencies between a client, an off-path attacker and a resolver

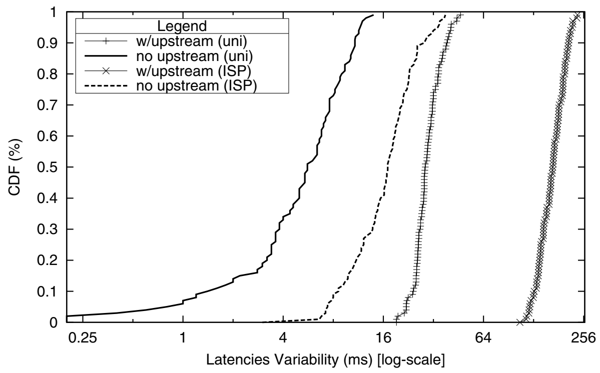


Fig. 2. Distribution of 100 measurements of δ values, with and without upstream resolver, for the two configurations: university proxy (with or without Google's upstream resolver), and an ADSL-connected proxy in an ISP (with or without the ISP's upstream resolver)

server (via both proxy and upstream resolver), can be broken down to three components, as shown in Figure 1: $\Delta_R = \Delta_{CP} + \delta + \Delta_{UA}$ (Δ_{CP} is the delay from client to proxy, δ is the delay from proxy to upstream resolver, and Δ_{UA} is the delay from the upstream resolver to the attacker's name server). We chose a different notation for δ since this is not a measured value, but computed using measurements of the other values: $\delta = \Delta_R - \Delta_{CP} - \Delta_{UA}$. Furthermore, δ is exactly the indicator for the use of an upstream resolver; when an upstream resolver is used, $\delta >> 0$, while when an upstream resolver is not used, $\delta \approx 0$. See our experimental results in Fig. 2. The detection exploits the fact, that we can easily measure Δ_R, Δ_{CP} and Δ_{UA}, and then test if δ is significant, indicating existence of an upstream resolver; see [18] for details and techniques to sample the Δ_R, Δ_{CP} and Δ_{UA} parameters.

Evaluation Results. We tested our proxy detection method via active measurements which we collected on two different network topologies: (1) puppet and local DNS resolver were set up on our university's wired 100Gb/s Ethernet network, and Google-Public-DNS (at IP 8.8.8.8) was used as an upstream resolver, and (2) puppet and the DNS resolver were set up on an ADSL network of a commercial ISP, and the upstream resolver was set up on a different network segment in the same wireless network of the ISP. We ran two tests in each topology (total of four tests): *without* an upstream resolver and *with* an upstream resolver. During each of the four tests we collected 100 samples (of latency) for each of the following parameters: Δ_{CA}, Δ_{CP}, and Δ_{UA} (in milliseconds). Based on these values we then calculated 100 values of δ for each of the four sequences of test (plotted in Figure 2), as follows: Repeat for $i = 1...100$: (1) calculate average of $j = 5$ samples[4] selected at random from a 100 samples from (each of) Δ_{CA}, Δ_{UA} and Δ_{CP} respectively, and (2) then calculate the latencies difference:

$$\delta_i = \frac{1}{j} \cdot \sum_{k=0}^{j} \Big(\Delta_{CA} - \Delta_{UA} - \Delta_{CP} \Big)$$

As can be seen from the measurements, Figure 2, the latency differences between configurations, with and without an upstream resolver, are significant.

3 Proxy DNS Resolver IP Address Discovery

In this section we show techniques to discover the address of the proxy, allowing denial of service as well as for cache poisoning attacks on proxy resolvers.

The idea is: (1) to find the network address block of the puppet, and then (2) to traverse the address block, until the resolver is found. To traverse the network block we apply IP defragmentation-cache poisoning. We present defragmentation-cache poisoning of a single host in Section 3.1. Then, Section 3.2, we show how to apply defragmentation-cache poisoning for resolver's IP discovery.

Network Address Block. The attacker runs a `whois.net` tool[5] on the IP address of the client (on which the puppet is running) to find out the network address block allocated to that network. One of the IP addresses on that network block is the IP address of the victim proxy resolver (that the attacker wishes to attack). The IPs range is typically not large; we ran a `whois` tool on 100,000 top domains according to Alexa, and found, that 70% of the networks have less than 2^{15} IP addresses, see Figure 3.

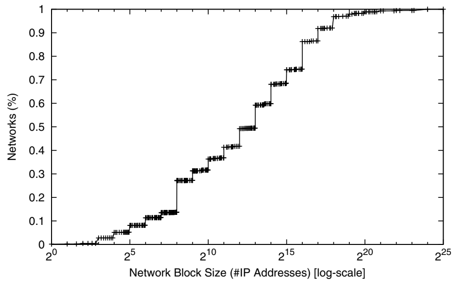


Fig. 3. The size of network blocks of 100,000 top domains from Alexa

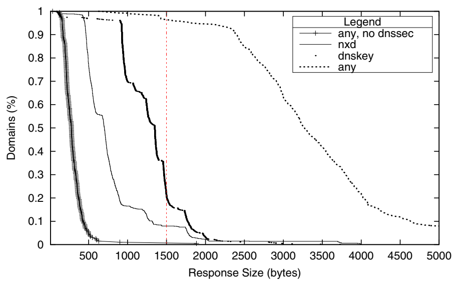


Fig. 4. Length of ANY, regular and NXDOMAIN responses of `gov` domains

[4] Other values of j could be used ($0 < j < 100$), but $j = 5$ provided sufficiently good results.

[5] `http://www.whois.net/ip-address-lookup/`

IP Fragmentation. The basic requirement of our IP address discovery technique is fragmentation: the attacker should (1) trigger a DNS request to a domain which responses exceed the MTU (maximal transmission unit), e.g., responses from domains that adopted DNSSEC typically exceed the MTU (see Fig. 4), and then (2) replace the authentic second fragment, of a fragmented DNS response, with a spoofed second fragment. The resulting packet is discarded by the resolver, and resolver retransmits the DNS request. If the packet arrives at host other than the resolver, no packet loss occurs. We use this timing channel to detect the IP address of the resolver.

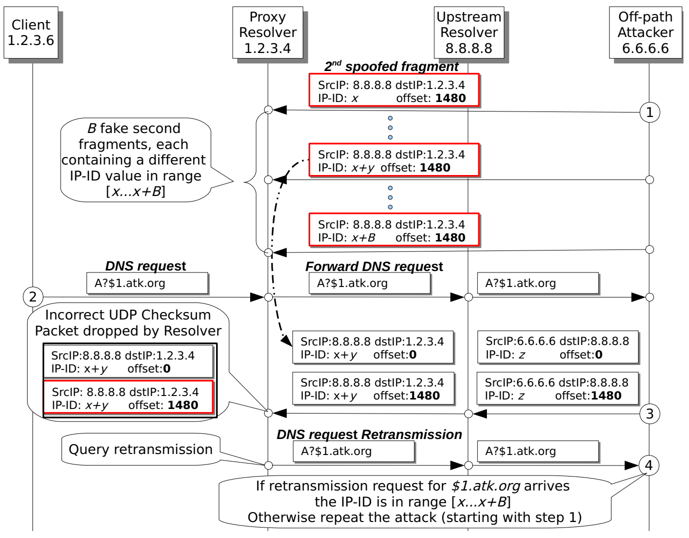

Client 1.2.3.6
Proxy Resolver 1.2.3.4
Upstream Resolver 8.8.8.8
Off-path Attacker 6.6.6.6
2nd spoofed fragment
SrcIP: 8.8.8.8 dstIP:1.2.3.4 IP-ID: x offset: 1480
SrcIP: 8.8.8.8 dstIP:1.2.3.4 IP-ID: x+y offset: 1480
SrcIP: 8.8.8.8 dstIP:1.2.3.4 IP-ID: x+B offset: 1480
B fake second fragments, each containing a different IP-ID value in range [x...x+B]
DNS request
A?$1.atk.org
Forward DNS request
A?$1.atk.org
A?$1.atk.org
Incorrect UDP Checksum Packet dropped by Resolver
SrcIP:8.8.8.8 dstIP:1.2.3.4 IP-ID: x+y offset:0
SrcIP: 8.8.8.8 dstIP:1.2.3.4 IP-ID: x+y offset: 1480
SrcIP:8.8.8.8 dstIP:1.2.3.4 IP-ID: x+y offset:0
SrcIP:8.8.8.8 dstIP:1.2.3.4 IP-ID: x+y offset:1480
SrcIP:6.6.6.6 dstIP:8.8.8.8 IP-ID: z offset:0
SrcIP:6.6.6.6 dstIP:8.8.8.8 IP-ID: z offset:1480
Query retransmission
DNS request Retransmission
A?$1.atk.org
A?$1.atk.org
If retransmission request for $1.atk.org arrives the IP-ID is in range [x...x+B] Otherwise repeat the attack (starting with step 1)
1
2
3
4

Fig. 5. Defragmentation-cache poisoning via a spoofed second fragment

3.1 Defragmentation-Cache Poisoning via Second Fragment

Defragmentation-cache poisoning is caching of spoofed fragments, in IP defragmentation cache, which get reassembled with the authentic fragments. The number of fragments that the recipient host can cache is limited[6]. We use B to denote the number of spoofed fragments sent by the attacker. The defragmentation-cache poisoning attack, illustrated in Figure 5, begins when the attacker sends B spoofed second fragments (step 1), which are stored at the defragmentation cache of the destination (for 30 seconds by default), and triggers (via a puppet) a DNS request, for its own domain, e.g., `atk.org` (step 2). When the authentic first fragment (of the DNS response) arrives, it is reassembled with the (cached) spoofed second fragment (step 3); the resulting IP packet has incorrect checksum, and is discarded by

[6] Typical defragmentation cache size allows several thousands of fragments; operating systems often impose a limit on the number of cached fragments per each (source, destination, protocol) triple. For example, in recent versions of the Linux kernel, the default value is 64 (and it is kept via the variable *ipfrag_max_dist*; see [19]).

the IP layer at the proxy (after defragmentation). Hence, the proxy retransmits the request after a timeout.

The probability that the IP-ID of a legitimate (fragmented) response matches the IP-ID of one of the (up to B) spoofed second fragments, which the attacker sent, depends on the IP-ID assignment method. We next analyse efficiency of defragmentation-cache poisoning for common IP-ID allocation methods: *incrementing* (supported by more than 70% of name servers) and *random* (supported by less than 1% of name servers); statistics are based on the IP-ID allocation methods supported by name servers of top level domains.

Random IP-ID. In a random IP-ID allocation the name server selects the IP-ID values in each response uniformly. Let n be the number of DNS requests triggered by the attacker and B the number of spoofed second fragments sent by the attacker. Note that defragmentation-cache poisoning allows to circumvent the *birthday protection*, thus enabling the attacker to trigger concurrent requests; see [18]. The probability for successful poisoning is:

$$\Pr[success] \cong 1 - \left(1 - \frac{B}{2^{16}}\right)^n \qquad (1)$$

See graph representing defragmentation cache-poisoning success probability, based on Eq. (1), in Figure 6; results of the experimental evaluation appear in the full paper version [18].

Incrementing IP-ID. Incrementing IP-ID can be either global (i.e., a single counter to all destinations) or per-destination (i.e., first IP-ID to some destination is selected at random and subsequent packets are allocated sequentially incrementing values). If the upstream resolver uses separate interfaces for communication to the Internet and for communication to the proxy, then the procedure for discovering the global and the per-destination IP-ID is similar. If the same network interface is used for communication to the Internet and to the proxy, then the IP-ID discovery, in case of a global counter, is simple: the attacker can trigger a query to a host that it controls and sample the IP-ID value. The attacker can efficiently hit the correct IP-ID by using a *meet-in-the-middle* strategy. The attacker triggers $\frac{2^{16}}{B}$ DNS requests and plants B spoofed second fragments in the defragmentation-cache of the recipient, each fragment i contains IP-ID value of $\{i \cdot \frac{2^{16}}{B}\}_{i=1}^{B}$.

3.2 Discovering Resolver's IP via Defragmentation-Cache Poisoning

The idea behind our IP discovery technique is the following: the attacker applies defragmentation-cache poisoning, via a second spoofed fragment, that is sent to each IP in the IP address block allocated to the victim network. The second fragment with the IP of the proxy poisons the defragmentation cache and ruins the DNS response sent by the upstream resolver to the proxy. The attacker then inspects the subsequent DNS requests from the upstream resolver to learn

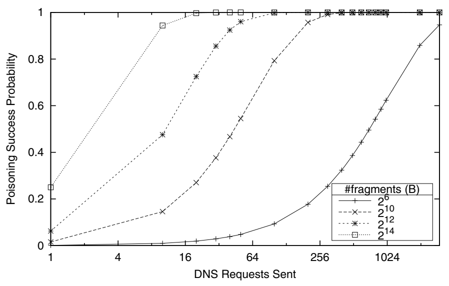


Fig. 6. Defragmentation-cache poisoning success probability per attempt, by analysis (Eq. (1)), for $B \in \{64, 1024, 4096, 16384\}$ (number of fake second fragments in cache) and different numbers of DNS requests, for random IP-ID assignment

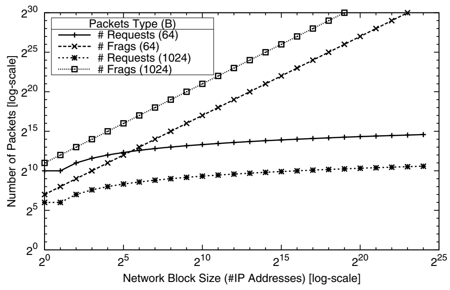


Fig. 7. Number of spoofed second fragments and DNS requests, for different network blocks ($2^0 - 2^{24}$), and for $B \in \{64, 1024\}$ (number of fake second fragments in cache), assuming per-dest IP-ID assignment

information about the reaction of the proxy. The reaction from the proxy is used as a side-channel, and allows to determine if the address of the proxy is found. Application of defragmentation cache poisoning for attacks is not new, and was mainly used for denial of service attacks, see [RFC6274] and [20,21]. The first application of fragmentation for attacks on DNS was in [15,16] for name server pinning and for cache-poisoning, respectively. Let B be the maximal number of fragments that can be cached at the defragmentation cache of the resolver, then $\frac{2^{16}}{B}$ is the maximal number of requests required in order to find the correct IP-ID for a single recipient, assuming a sequentially incrementing IP-ID is supported. Let $/x$ denote the CIDR subnet mask of the address block, returned from the `whois.net` query on the IP address of the puppet, and set $y = 32 - x$. Then 2^y is the number of IP addresses in a network block defined by the network part $/x$. We use binary search for proxy resolver IP address discovery, and since there are 2^y addresses, the procedure has to be repeated $\log 2^y = y$ times.

IP Address Discovery. For each $0 \leq i \leq y$, repeat:

(1) During attempt i the attacker triggers $\frac{2^{16}}{B}$ DNS requests (via the puppet) for records in its own domain, and sends $B\frac{2^y}{2^i} = B \cdot 2^{y-i}$ spoofed second fragments to a set of $\frac{2^y}{2^i}$ IP addresses (from the network address block); each fragment has a source IP address of the upstream resolver, destination IP addresses are from the network block (which the attacker traverses), and offset value of 1480. These fragments are stored at the defragmentation cache of the recipients, and then discarded if not reassembled (after 30 seconds by default).

(2) The upstream resolver receives the requests from the proxy and forwards them to attacker's name server. The responses to those requests result in (fragmented) referrals to subdomains of attacker's domain. The attacker inspects the subsequent requests from the upstream resolver to learn the reaction of the proxy resolver, and uses it as side-channel, to determine if the IP of the proxy was among the set of $\frac{2^y}{2^i}$ IP addresses that it sent the spoofed fragments to, in step (1).

(3) If the attacker sent the fragments to the correct IP address of the proxy, then when the authentic first fragment, sent by the upstream resolver, arrives, real first fragment and fake second fragment will be defragmented. Prior to accepting and caching responses, the resolver validates a number of fields, e.g., UDP port and checksum, DNS TXID. The reassembled response has incorrect checksum and is hence discarded by the proxy. If there is a pending DNS request, which the proxy resolver sent earlier to the upstream resolver, it times-out and the proxy retransmits its DNS request. The upstream resolver forwards[7] it to attacker's name server.

If the attacker receives a retransmitted DNS request - it knows that the IP of the proxy is in the set of $\frac{2^y}{2^i}$ tested IP addresses. In contrast, if the response arrives correctly to the resolver - there is no timeout, and the attacker receives a referral request (it responded with that referral to the request from upstream resolver in step 2). In this case, the attacker knows that the IP was not among the set of the IPs sampled in the current attempt, and repeats the attack with the next set of IPs.

Analysis and Experimental Evaluation. In our experimental evaluation the upstream resolver ran on a Linux OS, which implements a per-destination incrementing IP-ID allocation method, and our analysis and evaluation are adapted to this case; globally incrementing IP-ID can be sampled directly.

During the i^{th} iteration the attacker sends $\frac{2^{16}}{B}$ DNS requests and $B\frac{2^y}{2^i} = B \cdot 2^{y-i}$ spoofed second fragments ($0 \leq i \leq y$). The number of *spoofed second fragments* that the attacker has to send in the worst case, can be expressed via geometric series:

$$B \cdot 2^y \sum_{i=0}^{y} \frac{1}{2^i} = B \cdot 2^y \cdot \left(1 + \frac{1}{1 - \frac{1}{2^y}}\right) = B \cdot 2^y \cdot 2 = B \cdot 2^{y+1}$$

Notice that $\frac{1}{1-\frac{1}{2^y}} \approx 1$ since $\frac{1}{2^y} \approx 0$. The number of *DNS requests* that the puppet has to trigger in the worst case is: $\log 2^y \cdot \frac{2^{16}}{B} = y \cdot \frac{2^{16}}{B}$. The number of packets (requests and spoofed fragments) can be expressed as (see analysis in Fig. 7): $\left(\frac{2^{16}}{B} + B\frac{2^y}{2^0}\right) + ... + \left(\frac{2^{16}}{B} + B\frac{2^y}{2^y}\right) = y \cdot \frac{2^{16}}{B} + B \cdot 2^{y+1}$

Resolvers Behind NAT Devices. If the puppet and the proxy resolver are behind a many-to-one (network address translation) NAT device, then they share the same IP address. To take this possibility into account the attacker should start the search with the IP of the puppet, and extend its search at each iteration (following the binary search technique). We ran statistics on two CAIDA datasets from 2012 [9], that were collected on equinix-chicago and equinix-sanjose monitors on high-speed Internet backbone links. Both traces contained packets sent from distinct 89750 source IP addresses, collected over two minutes interval.

[7] To ensure that the upstream resolver does not respond to the proxy from the cache, the attacker sets a low TTL (time-to-live), e.g., TTL=0, on the requested record.

We ran the following test to check for DNS resolvers behind NAT devices: (1) we collected all DNS requests, i.e., packets sent to port 53; (2) we created a set of IP addresses that sent at least one DNS request per second (to ensure that we do not mistakenly interpret a host for a resolver behind a NAT; (3) we ran over the traces to check if those IP addresses also sent packets to other ports, including port 80, and 443. We then concluded with high probability that those resolvers were behind NAT devices. We came up with a total of 3492, out of 89750, resolvers behind NAT devices.

Restricted Rate. The attacker may be restricted to transmit at a low rate, e.g., to launch a stealth attack in order to evade detection in networks that are known to be well-monitored, or if the attacker does not have sufficient resources and its transmission rate is limited. We next calculate the number of IPs that the attacker can try at each attempt, when it is restricted by a rate of R Bytes/sec. Let τ seconds be the maximal time that fragments are stored in the defragmentation-cache; typical (default) value of τ is 30 seconds. Let f be the size (in bytes) of the second fragment; the second fragment can be of minimal size, e.g., 8 bytes and the 20 byte IP header. Then, the maximal number of IPs that the attacker can sample during a single attempt is $x < \frac{\tau \cdot R}{B \cdot f}$. With a modest rate of $50,000$ Bytes/sec (50 KB/sec), and $B = 64$ (fragments per host IP), the attacker can sample 781 IP addresses in a single iteration, and with 50MB/sec the attacker can sample at most $78,125$ IP addresses each time, i.e., more than the size of many network blocks. Most network blocks are not too large, and can be traversed efficiently (see Section 3.2 and Figure 3).

IP-ID Discovery. Once the attacker completes the IP discovery, it knows the that the IP-ID is in range of $\frac{2^{16}}{B}$ potential values. The attacker can also find the precise value of the IP-ID of the upstream resolver (in its communication to the victim proxy resolver); the knowledge of the IP-ID value is useful for UDP port discovery (see Section 4). The attacker again applies a binary search on the range of $\frac{2^{16}}{B}$ potential IP-ID values. This requires $\log \frac{2^{16}}{B}$ attempts in the worst-case. Since during each attempt the attacker sends $\frac{2^{16}}{B}/2^i = \frac{2^{16}}{B \cdot 2^i}$ spoofed second fragments, in the worst-case the attacker will send a total of $\frac{2^{17}}{B}$ spoofed second fragments:

$$\frac{2^{16}}{B} \sum_{i=0}^{\frac{2^{16}}{B}} \frac{1}{2^i} = \frac{2^{16}}{B} \cdot \left(1 + \frac{1}{1 - \frac{1}{\frac{2^{16}}{B}}}\right) \cong \frac{2^{17}}{B}$$

4 UDP Port Discovery and DNS-Cache Poisoning

The next step towards a successful cache poisoning is to find the port that the proxy resolver assigns to the request which the attacker wishes to poison.

We collected statistics from two CAIDA datasets from 2012 [9] and found that many proxies, which delegate DNS resolution to upstream resolvers, support the following popular port allocations: *fixed port*, *globally incrementing* and *per-destination incrementing*[8]. We used the traces to collect all the DNS requests (destination port 53) over UDP, and then filtered out IP addresses with a single DNS request, and collected only the sources that sent two or more requests. This allowed us to infer information about the source port allocation of the remaining DNS requests. We found that 30% of the requests were sent from some fixed port and 54% of the requests were sent from incrementing ports. Notice that the packets' traces are collected by CAIDA on (several) *backbone (OC192) links*, therefore, most DNS requests, appearing in those traces, are probably sent from proxies to upstream resolvers, since local DNS (proxy) resolvers are located on LANs; this premise is also coherent with the standard, [RFC5625] that states: 'proxy resolvers receive DNS requests from clients on the LAN side, forward those verbatim to one of the known upstream recursive resolvers on the WAN'.

The use of a fixed client port was shown to be vulnerable by Kaminsky [4]: the attacker triggers a DNS request to a name server under its control and learns the port the resolver uses for DNS requests. Security experts also identified the globally incrementing port assignment as vulnerable: the attacker can use a sampling procedure similar to [4], to obtain the current port value and then extrapolate the port that will be assigned by the (victim) resolver to the subsequent DNS request which the attacker wishes to poison. A per-destination incrementing port is believed to be secure, and is a recommended standard [RFC6056], since different ports' sequences are assigned by the resolver to different destinations; in particular learning the port value to one destination does not leak the port value to some other destination. We checked the *predictability rate* assigned by the popular DNS checker service provided by the OARC [7], to resolvers that send DNS requests with per-destination incrementing port. The tool reported (the highest) GREAT score to a *per-destination fixed* port (i.e., a different *fixed* port is assigned to each destination) and to a *per-destination incrementing* port (i.e., *sequentially* incrementing to each destination), indicating that both port allocation methods are believed to be secure by the DNS experts. However, our results (within) show otherwise.

In this section we present techniques that allow to predict the ports efficiently for each of the three popular allocation methods (above), contrary to folklore belief that, when a resolver does not send queries to the Internet directly, but only via an upstream resolver, it is secure. Our techniques do not rely on sampling the port, since in our setting this is not possible: the attacker *does not receive DNS requests from the proxy-resolver directly*, but only via an upstream DNS resolver. Furthermore, our results show that sequential allocation, whether per-destination or global, *surprisingly* allows for a more efficient port prediction, than a fixed port allocation; see comparison in [18].

During the port discovery the attacker triggers queries to a domain that it controls. This allows the attacker to control, not only the time at which the

[8] The effect of globally incrementing and per-destination incrementing ports' assignment methods is identical when proxies delegate DNS resolution to an upstream resolver.

request is triggered but also the *time at which the response is sent*; furthermore, if the attacker does not respond at all, this will result in a timeout at the resolver and in a retransmission of the DNS request. The attacker then traverses the port range until a correct port is found. Notice that often not all ports are used[9] and the supported ports ranges are significantly smaller than 2^{16}, e.g., it is considered safe to use ports in the range $(1024-49152)$, [RFC5452]. Therefore, some ports are more probable than others.

The attacker succeeds in a poisoning attack (of its own domain) when a correct port is found. As a result the attacker receives a subsequent request to the IP that was returned in the *poisoned record*, instead of a retransmission request.

However, prior to accepting and caching a DNS response resolvers validate a number of fields (recommended in [RFC5452]), e.g., IP addresses, UDP port, DNS TXID. The attacker knows the IP addresses: the address of the proxy-resolver was found using techniques in Section 3, and the address of the upstream is known since it sends the DNS requests to the name servers. The attacker has to find the correct UDP port and DNS TXID. A naive strategy is to apply the Kaminsky attack [4], however, this requires sending 2^{32} packets in the worst case in *each poisoning attempt* and is thus not feasible.

We devise a new approach for port discovery (explained next) which we dub the *Midway Rendezvous*. We show that this approach allows to significantly reduce the complexity of port discovery. We then propose to apply the midway rendezvous with two different strategies for port discovery: (1) an *optimised exhaustive search* and (2) *search via defragmentation-cache poisoning*; we compare the efficiency and complexity of both strategies in [18].

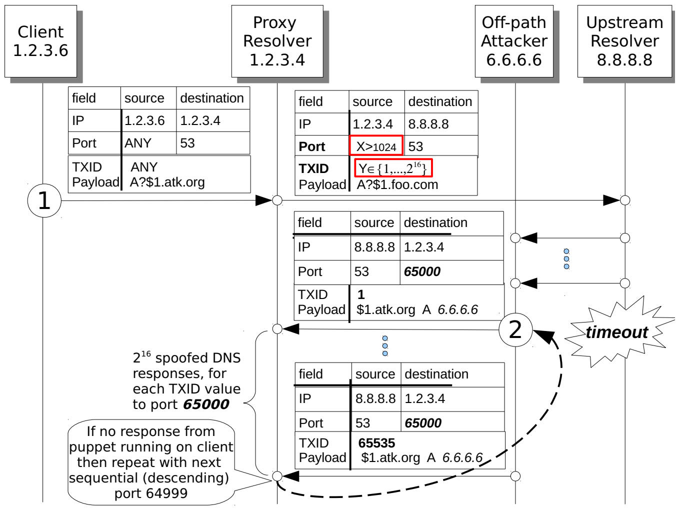
Midway Rendezvous. The idea is to traverse the ports range in a direction opposite to port incrementation, supported by the resolver. At each iteration i the attacker sends spoofed DNS responses, to p ports, each time decreasing the port number; p can be arbitrary, e.g., $p = 1$, and typically depends on attacker's bandwidth. Thus the attacker *walks* the port range towards the *direction* in which the resolver *walks*. In the worst case, they meet after $\frac{2^{16}}{2p}$ attempts, where p is the number of ports tried during each attempt. The value of p depends on port assignment method supported by the resolver. When incrementing ports are used, $1 \leq p \leq 15$; if the attacker samples a single port each time, i.e., $p = 1$, the attacker has to traverse half the ports range (assuming maximal ports range of 2^{16}). When a *fixed port* assignment is supported, $p = 0$, the attacker has to repeat the attack till it meets the fixed port (used by the resolver), and has to traverse in the worst case, $2^{16} - 1$ ports.

When next show how to apply this strategy using two different techniques *optimised exhaustive search* and *search via defragmentation-cache poisoning*, and compare efficiency.

[9] DNS running on Windows server 2008 uses ports range $(49152-65535)$ and Windows 2000/XP/server 2003 use ports from range $(1025 - 5000)$. Older Bind versions use fixed ports.

4.1 Optimised Exhaustive Port Search

The attacker applies the *midway rendezvous* to discover the port and proceeds as follows (Figure 8): For $i = 1...15$ or till 'port is found', repeat: (1) attacker triggers a DNS request to a record in a domain under its control, and (2) sends $p \cdot 2^{16}$ DNS responses to p (decreasing) ports' values starting with the highest port, e.g., 65535, for each possible TXID value (this is required to be able to detect when the correct port is hit, otherwise the response is discarded by the proxy resolver); in the worst case, the attacker sends $p \cdot 2^{16}$ responses. If the port is not one of the p ports tested at the current iteration, then increment i and update the port for next iteration. Although practical, this technique has a disadvantage: in order to hit the correct UDP port the attacker has to also guess the TXID. In the next section we show that the attacker can apply first-fragment defragmentation-cache poisoning to *split* the distribution of TXID and port to two separate distributions of size (at most) 2^{16} each (assuming all maximal number of ports is used).


Client
1.2.3.6
Proxy
Resolver
1.2.3.4
Off-path
Attacker
6.6.6.6
Upstream
Resolver
8.8.8.8
1
2
timeout
2^{16} spoofed DNS responses, for each TXID value to port ***65000***
If no response from puppet running on client then repeat with next sequential (descending) port 64999

field	source	destination
IP	1.2.3.6	1.2.3.4
Port	ANY	53
TXID	ANY	
Payload	A?$1.atk.org	

field	source	destination
IP	1.2.3.4	8.8.8.8
Port	$X > 1024$	53
TXID	$Y \in \{1,...,2^{16}\}$	
Payload	A?$1.foo.com	

field	source	destination
IP	8.8.8.8	1.2.3.4
Port	53	***65000***
TXID	**1**	
Payload	$1.atk.org A *6.6.6.6*	

field	source	destination
IP	8.8.8.8	1.2.3.4
Port	53	***65000***
TXID	**65535**	
Payload	$1.atk.org A *6.6.6.6*	

Fig. 8. DNS request port discovery: in step 1, the puppet triggers a DNS request to a resource within the attacker's domain. The off-path attacker, in step 2, at IP 6.6.6.6, sends 2^{16} fake responses (each containing a different TXID value) to each port of the DNS resolver. If failure - repeat the attack from step 2. When timeout, repeat the attack from step 1.

4.2 Port Discovery via First Fragment Defragmentation-Poisoning

The attacker can often improve the efficiency of port discovery, and in what follows we present port discovery which uses a technique we dub first-fragment defragmentation-cache poisoning. The steps of the attack are illustrated in Figure 9. We assume that the attacker knows the IP-ID value, e.g., it ran earlier the IP discovery phase, which also exposes the current IP-ID value (details in Section 3).

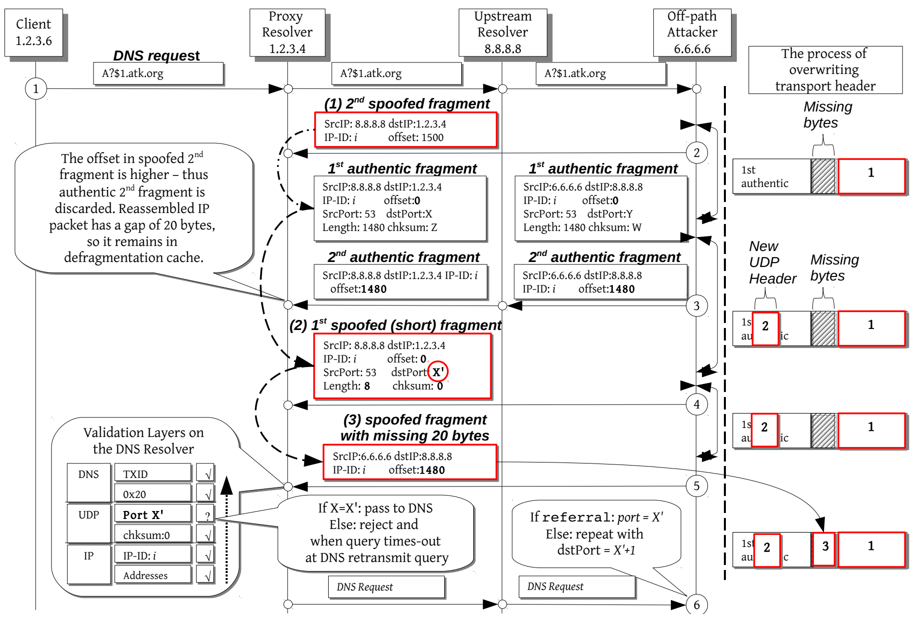

Client 1.2.3.6
Proxy Resolver 1.2.3.4
Upstream Resolver 8.8.8.8
Off-path Attacker 6.6.6.6
The process of overwriting transport header
DNS request
A?$1.atk.org
A?$1.atk.org
A?$1.atk.org
1
(1) 2nd spoofed fragment
SrcIP: 8.8.8.8 dstIP:1.2.3.4
IP-ID: i offset: 1500
2
Missing bytes
1st authentic
1
1st authentic fragment
SrcIP:8.8.8.8 dstIP:1.2.3.4
IP-ID: i offset:0
SrcPort: 53 dstPort:X
Length: 1480 chksum: Z
1st authentic fragment
SrcIP:6.6.6.6 dstIP:8.8.8.8
IP-ID: i offset:0
SrcPort: 53 dstPort:Y
Length: 1480 chksum: W
The offset in spoofed 2nd fragment is higher – thus authentic 2nd fragment is discarded. Reassembled IP packet has a gap of 20 bytes, so it remains in defragmentation cache.
2nd authentic fragment
SrcIP:8.8.8.8 dstIP:1.2.3.4 IP-ID: i
offset:1480
2nd authentic fragment
SrcIP:6.6.6.6 dstIP:8.8.8.8
IP-ID: i offset:1480
3
New UDP Header
Missing bytes
2
1
(2) 1st spoofed (short) fragment
SrcIP: 8.8.8.8 dstIP:1.2.3.4
IP-ID: i offset: 0
SrcPort: 53 dstPort: X'
Length: 8 chksum: 0
4
2
1
(3) spoofed fragment with missing 20 bytes
SrcIP:6.6.6.6 dstIP:8.8.8.8
IP-ID: i offset:1480
Validation Layers on the DNS Resolver
DNS
TXID √
0x20 √
UDP
Port X' ?
chksum:0 √
IP
IP-ID: i √
Addresses √
5
If X=X': pass to DNS
Else: reject and when query times-out at DNS retransmit query
If referral: port = X'
Else: repeat with dstPort = X'+1
2
3
1
DNS Request
DNS Request
6

Fig. 9. DNS port discovery: in step 1, the puppet triggers a DNS request to a resource within the attacker's domain. The off-path attacker, in step 2, at IP 6.6.6.6, plants a spoofed first fragment (with UDP length 8 bytes, and checksum 0) into the defragmentation cache of the resolver. The first authentic fragment is reassembled with the spoofed fragment, and then with the authentic second fragment. If the port is correct, the attacker will receive a referral, otherwise timeout and retransmission of the previous request.

The idea is to use fragmentation to overwrite the transport layer header of the fragmented IP packet sent by the upstream resolver to the proxy. In each such attempt the attacker sends a spoofed fragment with a source IP of the upstream resolver and includes a guess for a port. If the guess is correct - the response is accepted and cached by the resolver. Otherwise, if the port in the spoofed fragment is incorrect - the proxy rejects the response, and retransmits the request. This allows the attacker to distinguish the two events.

The goal of this step is to craft a spoofed first fragment, with a new port, and to overwrite only the transport layer header in the authentic first fragment. However, if two fragments contain identical offsets, then the last arriving fragment overwrites the first. Therefore, in order for the spoofed fragment to overwrite the transport header of the authentic fragment, it must arrive at the resolver *after* the first authentic fragment, and *before* the IP packet is reassembled, when the authentic second fragment arrives. Let $f = f_1||f_2$ be the IP packet consisting of two fragments f_1 starts at offset 0 and is of length $|f_1|$ and f_2 starts at offset $|f_1|$ and is of length $|f_2|$. The steps of the attack are described next.

(1) attacker sends a spoofed second fragment f_2', starting at offset $(|f_1| + \epsilon)$, where ϵ is some number of bytes.

(2) attacker triggers a DNS request (whose response is fragmented). When the first authentic fragment f_1 arrives it is reassembled with the spoofed second fragment f_2' that is already in the defragmentation cache; when the authentic second fragment f_2 arrives, it is discarded since a spoofed fragment starts and ends at a higher offset ($|f_1| + \epsilon$). However, the reassembled IP packet does not leave the defragmentation cache since there is a gap of ϵ bytes that are still missing.

(3) The attacker sends a short fragment that overwrites *only* the UDP header in the original first fragment. This fragment overlaps with first 8 bytes (the UDP header) with the authentic first fragment; the fragment contains `checksum` 0, which indicates that checksum validation is disabled[10], more fragments is set to 1 (`mf`=1), and `offset` is 0. When initiating the attack, the attacker sets the UDP port in this spoofed first fragment to 2^{16}, and decrements its value during each subsequent iteration, following the *midway rendezvous* strategy.

(4) Then the attacker sends a fragment that starts at offset $|f_1|$ and is of size ϵ to fill the gap.

4.3 Analysis and Experimental Evaluation

Let r be a DNS response size in bytes; for simplicity we round to 100 bytes (also in our experimental evaluation). Let R bytes/sec be the transmission rate of the attacker. Let t seconds be a limit on the timeout for a DNS request (i.e., including all retransmitted requests for that query) and let q be a number of times a pending query is retransmitted until it is terminated and SERVERFAIL is returned. Resolvers implement retransmission policy based on round trip time estimates of the name servers, [RFC1536], and support timeout management with exponential backoff. When a timeout occurs resolver enters an exponential backoff phase, i.e., the timeout is doubled, and query is retransmitted. Resolvers implement variable timeout and retransmission values, typically up to 45 seconds (which is also a recommended ceiling for total timeout for a query [RFC1536]), and attempt up to 15 retransmissions. For instance, Unbound*1.4.19* sets t to a maximal value of 40 seconds and Bind*9.8.1* sets $t \leq 30$ seconds and $q \leq 10$, i.e., supports up to 10 retransmissions before terminating a query.

In each retransmission the resolver advances the port (in case an incrementing allocation is supported). This allows the attacker to sample a number of ports in a single iteration (since with each retransmission there is a new pending request).

Optimised Exhaustive Port Search. The number of iterations i that the attacker has to repeat (or the number of queries that the puppet triggers) in the worst case, assuming that in a single iteration the resolver triggers q retransmissions (before terminating a DNS request) and the attacker samples p ports, is: (1) $i \leq \frac{2^{15}}{(p+q)}$ for incrementing port, and (2) $i \leq \frac{2^{16}}{p}$ for some (unknown) fixed port.

The maximal number of ports p that the attacker can test in a single iteration is $p \leq \frac{t \cdot R}{r \cdot 2^{16}}$; assuming that 2^{16} is the number of possible values of TXID.

[10] UDP checksum validation is optional, and it can be disabled by name servers by setting it to 0 (0000 in hexadecimal). When the checksum is disabled it is not validated by the resolvers.

The analysis vs evaluation results are in technical report [18]. Once the port is known the attacker launches a DNS cache poisoning attack, i.e., sends 2^{16} spoofed DNS responses, for some victim domain, such that each response contains a different TXID value.

Port Discovery via Defragmentation-Cache Poisoning. The number of iterations required to hit the correct port in the worst case is: $\frac{2^{16}}{(q+1)}$ for a fixed port and $\frac{2^{15}}{(q+1)}$ for incrementing port assignment; during each iteration the attacker matches the original query and up to q retransmissions. Since the attacker does not need to match the TXID, at each iteration only 3 fragments are sent (more fragments will not improve the efficiency of the attack); this significantly reduces the complexity of the attack. However, note that, the attacker cannot sample more than a single port, for each DNS request, since the payload is taken only from the last fragment, therefore, $p = 1$.

The worst-case number of requests required to guess the port is $\frac{2^{15}}{(q+1)}$ for incrementing allocation and $\frac{2^{15}}{(q+1)}$ for a fixed allocation. During each iteration $3(q+1)$ fragments are sent, thus the worst case number of fragments is $3(q+1) \cdot \frac{2^{15}}{(q+1)} = 3 \cdot 2^{15}$ for incrementing allocation and $3(q+1) \cdot \frac{2^{16}}{(q+1)} = 3 \cdot 2^{16}$ for a fixed port.

5 Conclusions

We presented DNS poisoning attacks on proxy DNS resolvers, i.e., resolvers which use an upstream resolver. This attack is significant, since a large and growing number of networks use upstream resolvers (and hence are vulnerable), and prior to this work, a common belief was that this setting protects the proxy resolvers from poisoning and DoS attacks. This belief is also partially due to the fact that DNS resolver-testing services, report this DNS configuration as secure. It is therefore imperative that networks, operating proxies, adopt appropriate corrective defenses, as described in [18].

Acknowledgements. We thank the anonymous referees for their comments on the earlier version of this work. This research was supported by grant 1354/11 from the Israeli Science Foundation (ISF), and by the Ministry of Science and Technology, Israel. We are grateful for support for CAIDA's Internet Traces [9] that is provided by the National Science Foundation, the US Department of Homeland Security, and CAIDA Members.

References

1. Akamai: Enchanced DNS (eDNS) (April 2013), `http://www.akamai.com/html/solutions/enhanced_dns.html`
2. Gudmundsson, O., Crocker, S.D.: Observing DNSSEC Validation in the Wild. In: SATIN (March 2011)

3. Kaminsky, D.: Dan Kaminsky's Blog, http://dankaminsky.com/2008/07/21/130/
4. Kaminsky, D.: It's the End of the Cache As We Know It. In: Black Hat Conference (August 2008), http://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf
5. Dagon, D., Provos, N., Lee, C.P., Lee, W.: Corrupted DNS resolution paths: The rise of a malicious resolution authority. In: NDSS. The Internet Society (2008)
6. Gibson Research Corporation: DNS Nameserver Spoofability Test (2012), https://www.grc.com/dns/dns.htm
7. DNS-OARC: Domain Name System Operations Analysis and Research Center (2008), https://www.dns-oarc.net/oarc/services/porttest
8. Provos, N.: DNS Testing Image (July 2008), http://www.provos.org/index.php?/archives/43-DNS-Testing-Image.html
9. CAIDA: Anonymized Internet Traces 2012 Dataset (2012), http://www.caida.org/data/passive/passive_2012_dataset.xml
10. Antonatos, S., Akritidis, P., Lam, V.T., Anagnostakis, K.G.: Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. ACM Transactions on Information and System Security 12(2), 12:1–12:15 (2008)
11. Herzberg, A., Shulman, H.: Unilateral Antidotes to DNS Cache Poisoning. In: Rajarajan, M., Piper, F., Wang, H., Kesidis, G. (eds.) SecureComm 2011. LNICST, vol. 96, pp. 319–336. Springer, Heidelberg (2012)
12. Klein, A.: BIND 9 DNS cache poisoning. Report, Trusteer, Ltd., Israel (2007)
13. Vixie, P.: DNS and BIND security issues. In: Proceedings of the 5th Symposium on UNIX Security, pp. 209–216. USENIX Association, Berkeley (1995)
14. Bernstein, D.J.: DNS Forgery (November 2002), Internet publication at http://cr.yp.to/djbdns/forgery.html
15. Herzberg, A., Shulman, H.: Security of Patched DNS. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 271–288. Springer, Heidelberg (2012)
16. Herzberg, A., Shulman, H.: Fragmentation Considered Poisonous: or one-domain-to-rule-them-all.org. In: IEEE CNS 2013, The Conference on Communications and Network Security (2013)
17. Herzberg, A., Shulman, H.: Antidotes for DNS Poisoning by Off-Path Adversaries. In: International Conference on Availability, Reliability and Security (ARES), pp. 262–267. IEEE, IEEE Computer Society (2012)
18. Herzberg, A., Shulman, H.: Vulnerable Delegation of DNS Resolution. Technical Report 13-05, Bar Ilan University, Network security group (April 2013)
19. Kernel.org: Linux Kernel Documentation (2011), http://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt
20. Gilad, Y., Herzberg, A.: Fragmentation Considered Vulnerable: Blindly Intercepting and Discarding Fragments. In: Proc. USENIX Workshop on Offensive Technologies (August 2011)
21. Gont, F.: Security Implications of Predictable Fragment Identification Values. Internet-Draft of the IETF IPv6 maintenance Working Group (6man) (March 2012) (Expires September 30, 2012)