Temporal Lensing and its Application in Pulsing Denial of Service Attacks

Ryan Rasti
Mukul Murthy
Vern Paxson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-129
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-129.html

May 26, 2014

Temporal Lensing and its Application in Pulsing Denial of Service Attacks

Ryan Rasti
UC Berkeley

Mukul Murthy
UC Berkeley

Vern Paxson
UC Berkeley

Abstract

We introduce *temporal lensing*—a technique that concentrates a relatively low-bandwidth flood into a short, high-bandwidth pulse. By leveraging existing DNS infrastructure, we experimentally explore lensing and the properties of the pulses it creates. We also show how attackers can use lensing to achieve peak bandwidths more than an order of magnitude greater than their upload bandwidth. While formidable by itself in a pulsing DoS attack, we note how lensing can be compatibly combined with amplification attacks to potentially allow attackers to produce pulses with peak bandwidths orders of magnitude larger than their own.

1 Introduction

Denial of service (DoS) first gained significant attention more than a decade ago [13], but its essential features have largely remained unchanged. That is, DoS attacks have grown larger in scale (most notably with distributed denial of service, or DDoS), but have often retained the mindset of brute-force flooding.

However, there are certain variants on DoS that are more intelligent. Some exploit abstracted-away features of protocols. For example, SYN flooding attempts to mount a state-holding attack in implementations of TCP. In addition to protocols, certain types of DoS attacks also exploit existing infrastructure. Reflector attacks [15], for instance, trick innocent third parties into assisting. The reflector can serve the dual purpose of hiding the attacker's identity and increasing the efficacy of the attack, usually by reflecting a larger payload to the victim.

The plethora of protocols and deployed infrastructure—much of it originally designed with little regard for security—opens the door to exploitation. We introduce *temporal lensing*, which (a) exploits existing infrastructure (reflectors) in the novel way of *concentrating* a flood in time, and that (b) can be used to significantly increase the effectiveness of an attack on a feature of a widely deployed protocol (TCP congestion control). We experimentally validate lensing and note its ability to empower an attacker.

2 Related Work

Kuzmanovic and Knightly [8] first describe the concept of bursty, low average bandwidth pulses as "shrew" attacks. The attack aims to send enough packets in a short duration to cause a TCP RTO (retransmit timeout) in clients and then periodically induce another RTO with each pulse. They note that such attacks, due to their low average bandwidth, are harder to detect than traditional flooding. They show that such an attack is effective at reducing throughput by an order of magnitude or more. Also, they perform experimentation showing the drawbacks of two proposed defences. Specifically, RED requires longer measurements to correctly identify shrew pulses; increased randomization of RTOs involves a trade-off with throughput in the absence of an attack.

Luo and Chang [12] generalize the idea of shrew attacks from attacks on RTO to attacks on TCP congestion control in general. That is, they also consider an attack on TCP's AIMD (additive increase multiplicative decrease) mechanism and show that it can also severely degrade TCP flows. Guirguis et al. [5] further abstract such low average bandwidth attacks as a type of RoQ (reduction of quality) attack—that is, pulsing exploits transients in a system (congestion control for the example of pulsing), instead of its steady state capacity (victim's bandwidth). Their empirical analysis was limited to a system other than TCP and congestion control (they experiment on bursty web requests effects on system resource utilization and ability to serve other requests). However, they were able to show the theoretical and practical potential of RoQ attacks in general.

While pulsing DoS boasts impressive theoretical and experimental efficacy, it has seen little use in practice.

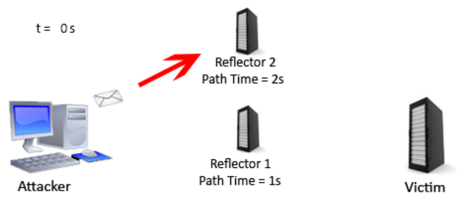
We suggest a reason to be as follows: senders are limited to their uplink bandwidth in a pulse, so simple pulsing cannot intuitively perform better than flooding in terms of damage inflicted. However, the idle time between pulses indicates room for improvement. We propose a method that allows an attacker to send in a flood, but concentrates this flood into pulses at the victim, allowing the attacker maximal bandwidth utilization.

Paxson [15] describes the role of reflectors in DoS attacks as amplifying the flooded traffic and helping attackers evade detection. He also notes the natural use of open DNS resolvers as reflectors. Our attack prototype takes advantage of this last fact and the abundance of such resolvers (estimated to be in the tens of millions [16]). However, we take a new angle on reflectors, instead using them to concentrate the arrival of packets at the victim, much like a lens focuses light. This method is orthogonal to traditional amplification attacks.

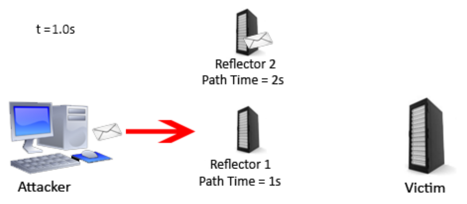
The proposed attack requires the ability to calculate latencies between arbitrary hosts, in our case between a reflector and the victim. The first major effort at latency estimation was IDMaps [4] which calculated distance between any two hosts using a map of network tracers. After IDMaps, many other systems emerged [14, 3, 18, 17]. However, dependence on new protocols or infrastructure limit these schemes applicability to our proposed attack.

These schemes note certain relevant pitfalls with estimating latencies. Namely, Ledlie et al. [9] discuss factors which cause differences between expected and actual latency between hosts. One unavoidable factor is that latencies can be skewed by certain anomalous data such as timeouts, which Ledlie et al. [10] remedy by using medians instead of means and employing various filters. We borrow some of these methods (specifically giving less weight to outliers in favor of medians and outlier-independent measures) when building our estimates of latencies.

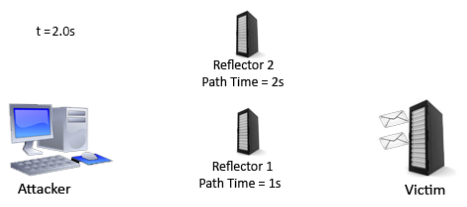
Another latency estimation system, Gummadi et al.'s King [6] offers the impressive advantage that it does not need additional architecture or cooperation beyond what is currently found in DNS. King works by finding DNS servers close to the end hosts, and estimating distances using recursive DNS queries. King is particularly well suited to our task because: (a) it requires no additional architecture or explicit agreement beyond standard DNS queries, (b) it is most accurate if either (or both) hosts are DNS servers, and (c) recursive protocols, such as DNS, naturally lend themselves to use in reflection. We also introduce an addition to King—by taking its cache poisoning trick a step further—allowing for a direct round-trip time computation between an open DNS resolver and an arbitrary host in Section 7.1.



(a) At $t = 0$, attacker sends one packet towards resolver 2



(b) At $t = 1$, the first packet is halfway along its path to the victim and the attacker sends another packet to resolver 1



(c) At $t = 2$, both packets arrive at the victim

Figure 1: Attack illustration. Paths through reflectors 1 and 2 have attack path latencies of 1 and 2 seconds respectively. The attacker sends at a rate of 1 pps, but he concentrates his flow such that two packets arrive in one second at the victim. For a brief moment, he has effectively concentrated his bandwidth two-fold.

3 The Attack

3.1 Motivation and Attack Description

A normal pulsing attack has one major place for improvement: a majority of the attacker's bandwidth is unused in between pulses. Therein lies the question of how to send packets during these idle times but have these packets still arrive at the victim within a pulse.

We draw an analogy to the military strategy "Time on Target" [7] to synchronize artillery fire. Using synchronized clocks and estimates of projectile flight times, a coordinated military can fire from different locations but have all their fire hit the target at the same time. In a more modern version of this idea, "Multiple Rounds Simultaneous Impact", a single artillery can make multiple rounds rendezvous at the target by varying the angle of fire, charge, and thus the flight time. By varying projectile paths, an artillery can make more shots arrive at the victim in one period of time than it can send in that period. We leverage the wide range of paths and latencies on the Internet to accomplish a similar feat. If an attacker can schedule sending in such a way that the attacker first sends packets that will take longer to arrive and then sends those that will take shorter to arrive, they can rendezvous within a small window of time.

However, if the attacker sends directly to the victim, the latency is constant. Every packet will take about the same amount of time to reach the victim since they travel along the same path.

Reflectors introduce the ability to have variable *attack path latencies*: the time from attacker through reflector to victim. Each reflector used potentially introduces a new path for attack traffic and thus a different attack path latency. A simple example is illustrated in Figure 1.

In the remainder of this paper, we will call this technique *temporal lensing* or simply *lensing*, as reflectors can temporally concentrate packets, much like a lens focuses light. Also, when describing how the attack works, we prefer the term *concentration* to amplification, as the former is more fitting and the latter is already used to describe an orthogonal attack.

3.2 Strategy

The actual attack can be decomposed into three main parts, which we develop in the next three sections of the paper:

- determining attack path latencies through resolvers to the victim (Section 4)
- building a sending schedule to create maximal lensing from these latencies (Section 5)
- conducting the attack (Section 6)

After experimentally validating lensing, we turn our attention to extensions to our basic attack (Section 7) and finally to defenses (Section 8).

4 Obtaining latencies on the Internet

To actually carry out the attack using some set of reflectors, we first need to know, for each reflector, the attack path latency. Estimating attacker to reflector and attacker to victim latencies are trivial—any sort of ping will suffice. We still need a way to measure latency from the reflector to victim.

Measuring latencies between two Internet end hosts, however, is a well studied problem [4, 14, 3, 18, 17]. We chose a particular method, King [6], that is particularly

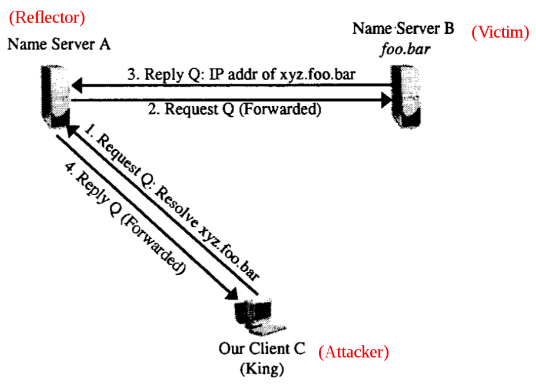


Figure 2: The operation of King (reprinted with approval from [6]), with the relevant actors for lensing added in red

well suited to our task. King operates by issuing recursive DNS queries between two DNS servers located close to the end servers in question. In Figure 2 we use a figure from King and overlay it with labels of the attacker, victim, and reflector, which are features of our attack. The figure shows how, with a single recursive query, an attacker can get an estimate for the attack path RTT, by taking the difference in time between when the attacker sends the query to when the attacker receives a response.

King must deal with two conflicting caching issues. First, it "primes" the resolver so that it caches the fact that the victim is authoritative for its domain. This prevents the resolver from iterating through the DNS hierarchy for a query. Second, the attacker must issue queries for different subdomains of the domain the victim (`foo.bar` in the example), lest the attacker hit the resolver's cache and short-circuit the query. King (and our attacker) bypasses this last issue by sending queries for random subdomains, each of which requires the entire chain of packets 1-4 in the figure to be sent.

So, for our attack, if we limit our reflectors to recursive DNS resolvers (which by their recursive nature can perform such reflection naturally), such estimates are made more accurate. If we further limit the victim to a DNS server[1] as well, then King accurately measures the attack path RTT, which can be halved to obtain the attack path latency. One may intuit that halving the RTT may give a one-way latency, but this by no means needs to be the case. Our positive experimental results on lensing and pulsing in Section 6, however, experimentally validate

[1]The reader may correctly note that pulsing DoS attacks (which attack TCP congestion control) will probably have little impact on a UDP based service with short quasi-flows such as DNS. We defer a discussion of estimating attack path latencies to TCP-based hosts to Section 7.1

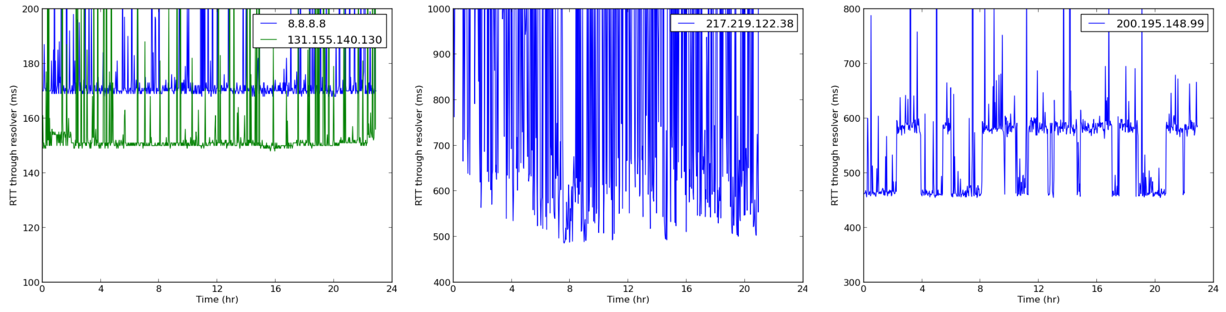


Figure 3: Two good resolvers (Google and Eindhoven University of Technology) with minimal path latency variation (roughly flat lines), an obviously bad resolver with high path latency variation (where 1000ms is a timeout), and a resolver that appears good over small samples of time but is actually bad for our attack, respectively. We took samples 2 minutes apart.

this heuristic.

In short, King provides a suitable method for estimating the attack path latency.

4.1 Open DNS Resolvers

We take a short detour to discuss relevant practical aspects of DNS.

[16] has put recent estimates of the number of open DNS resolvers in tens of millions, with many running on (often outdated) commodity hardware. Further it notes that many of such resolvers are ephemeral and last on a given IP on the order of days to weeks.

[16] also notes that many resolvers do not iterate the DNS hierarchy themselves, but instead forward the work off to auxiliary resolvers. TurboKing [11], an extension of King, explicitly acknowledges this possibility to improve its accuracy. However, since the traffic for our attack will follow the same path as that for determining latencies, such DNS nuances should not affect the attack path latency calculations.

4.2 Attack Path Latency Variation

4.2.1 Short-term Variation

One point of concern is how accurate attack path latency measurements are over a short period of time. In particular, it may take a few minutes to just measure latencies to all of the resolvers (as is the case for our prototype). We want to explore how attack path latencies vary over this period of time. Depending on how much they do, it may render just minutes-old estimates invalid.

To this end, we took a sample of 44 resolvers from a public list of about 3000 [1] and measured the path latency through each one once every two minutes.[2] Figure 3 shows selected examples of what the path latencies looked like over time in the cases of (from the attacker's point of view): good latency variation, bad variation, and deceptively good variation.

After taking a few samples over a few seconds, it would be obvious that the resolvers in the first graph are good[3] and the resolver in the second graph are not. However, the resolver in the third graph may also be considered good because it had few timeouts and a fairly consistent latency over a short period of time. But over longer periods of time, it looks as if there are routing changes that abruptly change the attack path latency; this could cause packets sent to this resolver to miss the pulsing window.

We found that the interquartile range (IQR, which is defined as the difference between the 75th and 25th percentiles) was an effective feature for identifying these misleading resolvers. Fortunately for an attacker, these misleading resolvers are also rare; while the one in that third graph had an IQR of 122 ms, Figure 4 shows that almost half our resolvers had an IQR of less than 12 ms.

So for some resolvers, the attacker must either perform latency measurements immediately before the attack or have a longer period of statistics to show that the resolver is not a good one and not use it. However, as Figure 4 shows, these types of resolvers are uncommon. So even if an attacker does not account for the cases of misleading resolvers and just assumes every resolver that appears good over a short period of time is indeed good, performance will not significantly suffer.

[2]We take appropriate measures (previously discussed) to make sure DNS caching does not invalidate our results.

[3]The spikes represent timeouts, which disproportionately distort the graph. For resolvers where they occur infrequently, we believe them to be the result of low (but non-zero) packet loss rates.

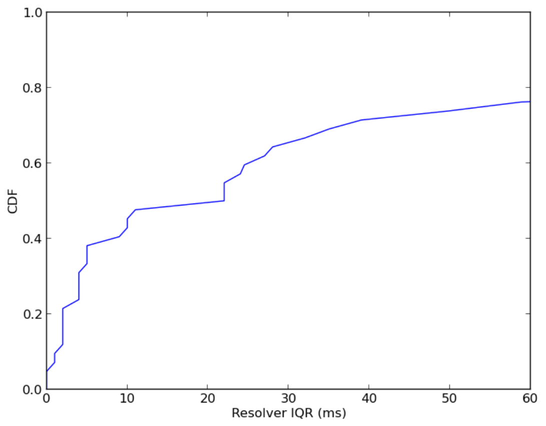


Figure 4: Cumulative Distribution Function of IQR ranges of path latencies for each resolver, with samples taken every 2 minutes over 24 hours.

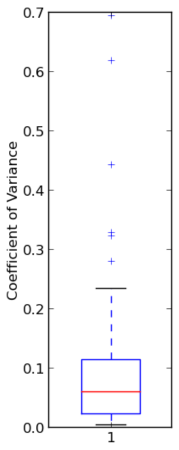


Figure 5: Box plot of the coefficients of variance of each resolver's set of medians. A low CV suggests that a resolver's path latency to the victim did not change much over the course of our 10-day period.

4.2.2 Long-term Variation and Caching

Since we have just discussed some challenges gaining accurate measurements of path latencies through different resolvers and declaring resolvers good, it makes sense to consider caching these results to save time and bandwidth computing them again later on. Note that before we were worried about how steady a resolver is over a period of minutes; now we are concerned about the validity of that measurement over the course of days or weeks.

To examine how the latencies change over longer periods of time, we took our same sample of resolvers and sent 50 packets (after taking care to warm-up NS record caching and bypass A record caching) through each resolver every 4 hours for about 10 days. For each resolver and each set of measurements, we took the median of the 50 latencies. For each resolver, we computed the standard deviation of these medians for each sample time (there were a total of 59 of these medians, assuming all measurements were successful for that resolver). Since we are interested in comparing variation, we divided the standard deviations by the means; these terms are called coefficients of variation and are normalized standard deviations. The distribution of these coefficients of variation are shown in Figure 5.

Since the data shows that some resolvers have very little variance over time, we believe that many resolvers' measurements can be cached. Resolvers with coefficients of variation below a certain threshold (we believe 0.02 is reasonable) have consistent path latencies, and their latency times can be cached. If we were to run the same statistics on a much larger group of resolvers - which would take excessive time and bandwidth - we would be able to identify a large set of these resolvers and save a shortlist of high-quality resolvers to speed up future attacks.

We also note that we did not find many resolvers with high attack path latencies. Having resolvers with high path latencies (with low variation) would enable us to increase our theoretical bandwidth gain because we would have a longer period of time to send packets over (and subsequently concentrate). However, we found that most high latency paths exhibited high jitter and inconsistency. As Figure 6 shows, none of the resolvers in our sample with more than 450 ms latency had a coefficient of variance below 0.02, and resolvers over 600 ms latency had about 0.1 coefficient of variance or greater. This makes sense because higher latency paths generally have more hops, which means more room for inconsistencies.

4.3 Amplification Attacks

Since we are using DNS resolvers as reflectors, it may occur to the reader that lensing can be combined with amplification. To study pulsing in isolation, however, we will concentrate solely on lensing and defer a discussion to how amplification might be added to the attack and its consequences to Section 7.2.

Lastly, we emphasize that DNS is simply our chosen vector for the attack because of: its simplicity in determining latencies, the availability of recursive DNS resolvers, and the fact that recursion allows reflection without IP spoofing. However, essentially any type of reflection is compatible with lensing.

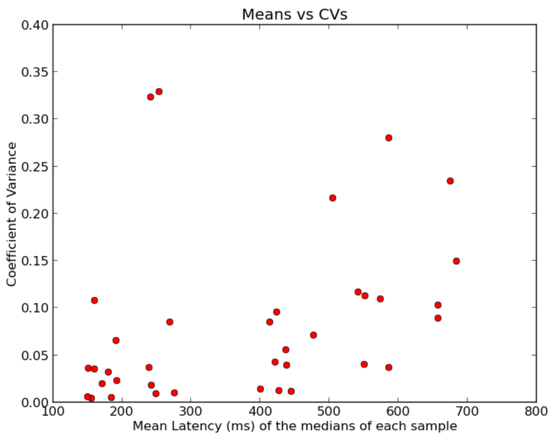


Figure 6: Plot of the average latency of each resolver against that resolver's coefficient of variance, as computed earlier. Three resolvers had CVs greater than 0.4 and are not shown.

5 Building an Optimal Schedule

Now that we have obtained path latency estimates for resolvers, we must compute a sending schedule. This schedule divides up the sending window into a set of time slots T and states which resolver an attacker should send to for each slot. The number of slots available for the attacker to send is a function of the attacker's maximum bandwidth and the range of path latencies measured for different resolvers.

We have some set of reflectors R and set of time slots to send T. Define the *pulse window* or simply *window* to be the duration of the pulse as seen at the victim. In trying to create the largest pulse, our goal is maximize the expected number of packets that land in a predetermined window. We use a greedy algorithm to compute the optimal schedule given an initial set of reflectors and estimates of their corresponding attack path latencies. According to our algorithm, at each time slot t, we simply choose the reflector which provides the highest estimated probability of landing within the window. This greedy algorithm is indeed optimal. We defer a proof to Section 12.

However, the problem statement and proof do not account for distortions in the attack path latencies due to the attack itself—such as those caused by effects of straining resolvers and congestion caused by the attack. Since the attack is distributed over geographically diverse resolvers, any congestion should decrease exponentially with the number of hops from the victim. So, it is not clear if congestion would actually be detrimental from an attacker's point of view. Further, we find little evidence of congestion that would actually inhibit an attack in our experimentation.

6 Attack Experimentation

Armed with the ability to estimate attack path latencies and to build an optimal schedule from them, we are finally ready to experiment with lensing.

6.1 Methodology

We simulate attacks on machines under our control. We use an Windows Azure VM instance on the West Coast as our attacker. We use another Azure VM instance and an Amazon Web Services VM instance, both on the East Coast, as our victims. We use a publicly available list of just over 3,000 resolvers [1] as our reflectors.

We register a domain name[4] that allows us to have an authoritative DNS server. We make the AWS victim instance authoritative for our domain and the Azure victim instance authoritative for a subdomain of the original. This allows us to send recursive DNS queries through any open recursive DNS server to either of our victims.

Before an attack, the attack machine quickly scans the resolver list. It issues recursive queries to the resolvers (just as in the attack) to obtain latency measurements.[5] For each resolver, it constructs a histogram for the distribution of each attack path latency. Then, it essentially performs the optimization algorithm in Section 5 with the histograms serving to compute probabilities. During the attack, we just send to the resolvers according to the schedule.

6.2 Features Measured

6.2.1 Dimensions

We measure pulsing on a few dimensions—specifically how our attack metrics vary as a function of:

- attacker bandwidth (to simulate attacker's with different resources, we throttle our sending appropriately)
- maximum bandwidth to each reflector (to see how this affects this attack, and to avoid overburdening or getting throttled by a resolver)
- pulse window size (the duration during which we are trying to have packets arrive at the victim)

[4]`pulsing.uni.me`

[5]We chose to take ten samples from each resolver, which turned out well for our attack

In addition, the attacker might have separate reasons own for throttling bandwidth to any given reflector (for example to avoid arousing suspicion). Also, since the resources of the reflectors we use are unknown, we err on the side of caution when sending to them. To any given one, we send at a maximum bandwidth of 500 pps over a course of 20-100 ms (which translates to a maximum of 5 KB).

We do not explicitly explore the number of reflectors used as a dimension, because the number of reflectors heavily depends on the existing dimensions of attacker bandwidth and maximum bandwidth to each reflector.

6.2.2 Metrics

In Section 5 we defined an optimal schedule as one that has the greatest expected amount of packets falling within the pulse window. This is intuitively a natural parameter to maximize. However, if we solely use *absolute* number of packets as our *metric* of efficacy, it will be artificially inflated just by increasing the uplink bandwidth of the attacker or increasing the target window size. So, we choose some additional bandwidth-agnostic metrics:

- *bandwidth gain*: $\frac{\text{bandwidth in the pulse window at the victim}}{\text{attacker's maximum sending bandwidth}}$
- *concentration efficiency*: $\frac{\text{\# packets landing in window}}{\text{\# total packets sent}}$

The first metric is the most important from the short-term point of view of the attacker. It gives the attacker a sense of how much extra bandwidth can be produced.

The second metric, however, provides a good basis for determining how the attack scales with the attacker's bandwidth. Because the size of the reflector pool is constant, as we increase sending (i.e., attacker) bandwidth, there will be more time slots when there will be no available resolver to send to (because we throttle bandwidth to any given reflector). In this case, the bandwidth gain will artificially decrease. To see this, imagine we send a maximum of one packet to each of 100 reflectors—then we can at most send 100 packets to the victim, regardless of our uplink bandwidth. However, all things equal, the concentration efficiency will remain constant.

These two metrics must be used in conjunction with each other, as it is easy to inflate one metric at the expense of the other. A large pulse window size would lead to concentration efficiency of 1 (ignoring packet drops) but no bandwidth gain. An ultra-small target window could result in an extremely high bandwidth gain (if one packet lands in it) but a very low concentration efficiency.

Lastly, we measure bandwidth in terms of packets per second. The packets we send and that are reflected are small (around 100 bytes), which may make the numbers look artificially high. For a sense of scale, 10K pps roughly translates to 8 Mbps.

6.3 Results

Figure 7 shows the results of a pulse simulating attackers with different bandwidths at a fairly thin pulse window of 20 ms. The simulation artificially caps our outgoing bandwidth by appropriately adjusting the minimum time between which any two packets can be sent. The bandwidth gain can essentially be calculated by dividing the height of the pulse bucket by that of the tallest bucket for the attacker's sending. For the low-bandwidth case, we see a gain of slightly over 14 times for low bandwidth, 10 times for the moderate bandwidth, and 5 times for the high bandwidth. The efficiency can be calculated by dividing the area of the pulse bucket by that of all the buckets on the left (sending) graph. The efficiency is around 50% for the low and medium cases and just under 40% for the high bandwidth case.

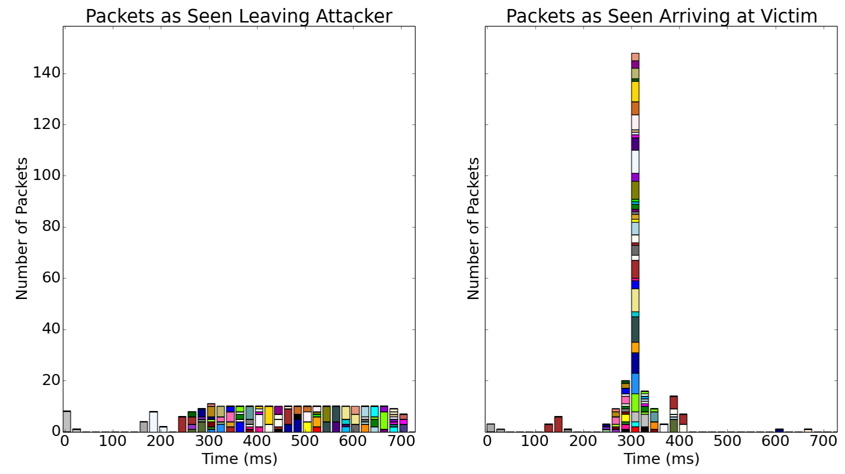
The colors on Figure 7a map onto the reflectors used—we can see that a number of reflectors contribute to the pulse and some do not at all (because their latency measurements were off).

We see what look like multiple pulses on Figures 7b and 7c. The packet traces reveal that these secondary spikes are retransmits by the resolvers. The victim (an authoritative DNS server) was not able to keep up with the rate of queries and failed to respond to many of them. The resolvers then timeout and retransmit; since many of them share a common retransmit timeout, their retransmits rendezvous at time = (original pulse time) + (retransmit timeout). We were able to determine, then, two common retransmit timeouts of 800 ms and 2 s. We discuss some ways an attacker may be able to leverage retransmits in section 7.4.2. Another feature of retransmits is that the total number of packets received by the victim often exceeds the total number sent by the attacker by about a factor of two.[6]

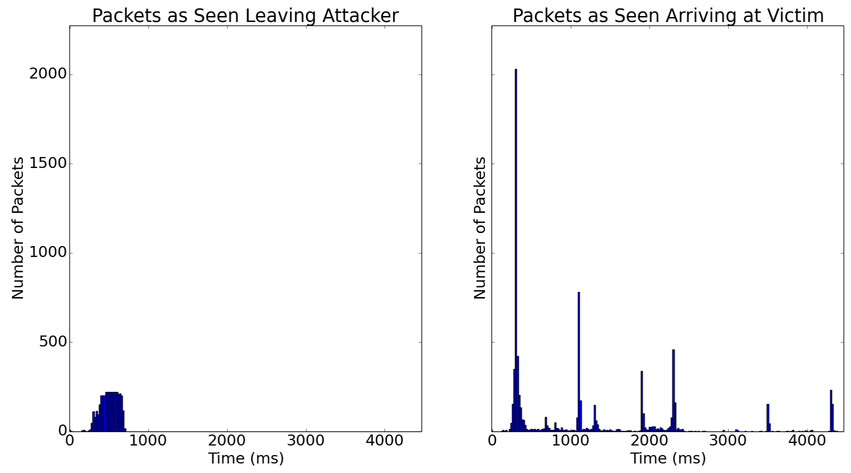
Figure 8 shows how the lensing metrics vary with pulse window duration (the dimensions attacker bandwidth 10K pps, max bandwidth to reflector of 500 pps are fixed). Bandwidth gain (and absolute pulse bandwidth) are expected to fall as window size increases because we are essentially spreading the pulse out. Efficiency sees a slight increase, but seems to level off at larger window sizes. We hypothesize that there are a number (about 40%) of reflectors that show an extremely high attack path latency variation but are nonetheless chosen as reflectors.

Figure 9 shows lensing properties as a function of maximum bandwidth to any reflector (dimensions attacker bandwidth 10K pps, window size 20 ms are fixed). The variation in the metrics of bandwidth gain and pulse bandwidth are not too illuminating, since they vary only

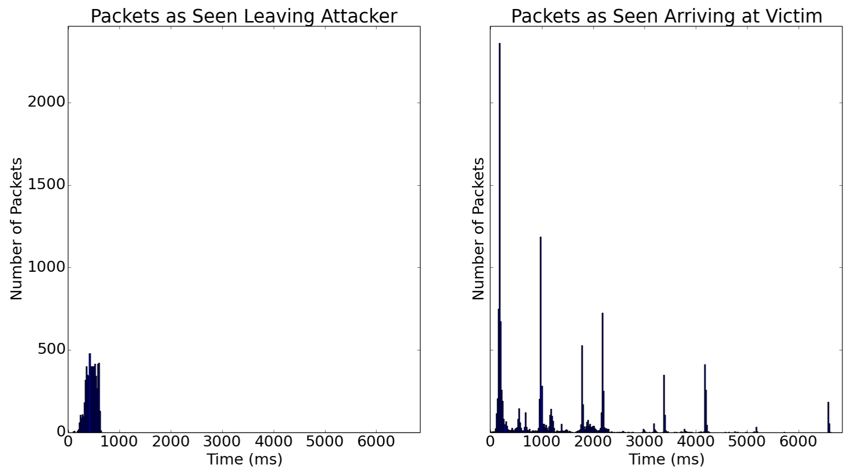
[6]But our metrics are carefully chosen so that retransmits do not affect them.



(a) Pulse from Low Bandwidth (500 pps) sending, 75 reflectors sent to

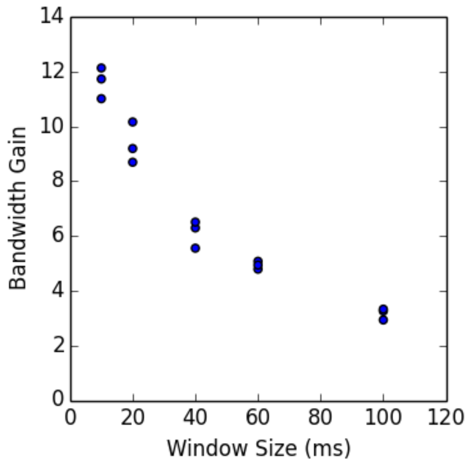


(b) Pulse from Medium Bandwidth (10,000 pps) sending, 816 reflectors sent to

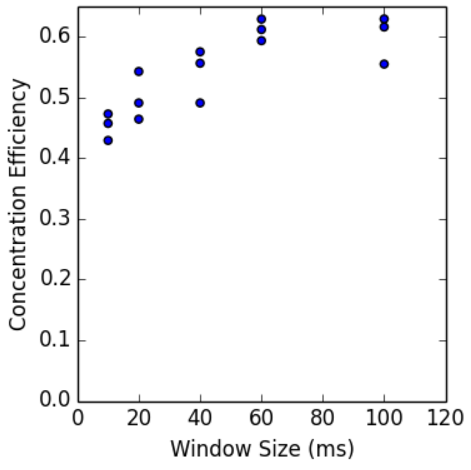


(c) Pulse from High Bandwidth (20,000 pps) sending, 1201 reflectors sent to

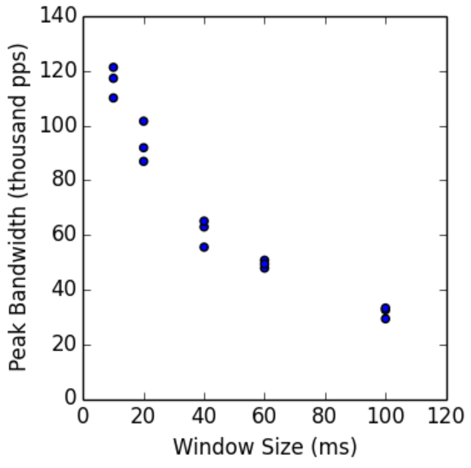
Figure 7: Selected Pulses performed on the AWS Instance (using dimensions of max bandwidth to reflector 500 pps, window size 20 ms). The bucket size is set equal to the pulse window size (20 ms).



(a) Bandwidth Gain

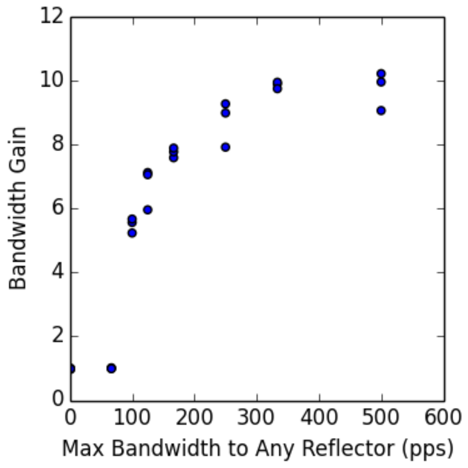


(b) Concentration Efficiency

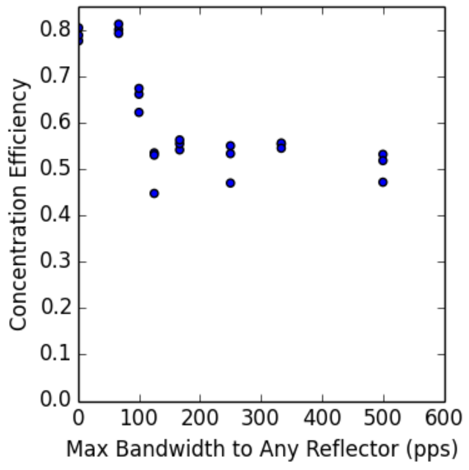


(c) Pulse Bandwidth

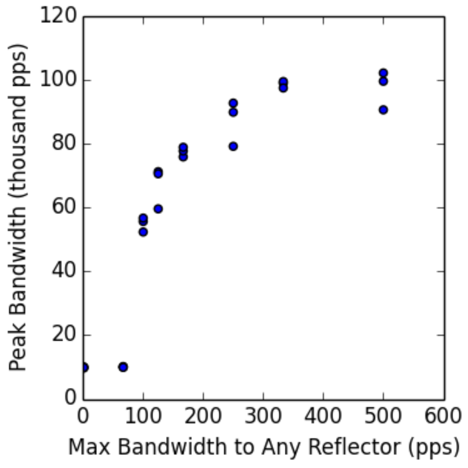
Figure 8: Lensing Metrics as a Function of the Desired *Pulse Window Size* (with the AWS instance as victim—fixed dimensions: attacker bandwidth 10K pps, max bandwidth to reflector of 500 pps)



(a) Bandwidth Gain

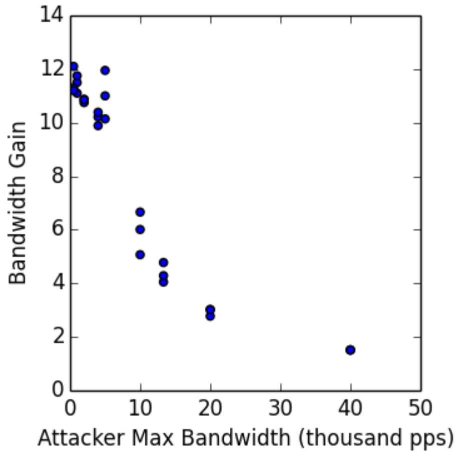


(b) Concentration Efficiency

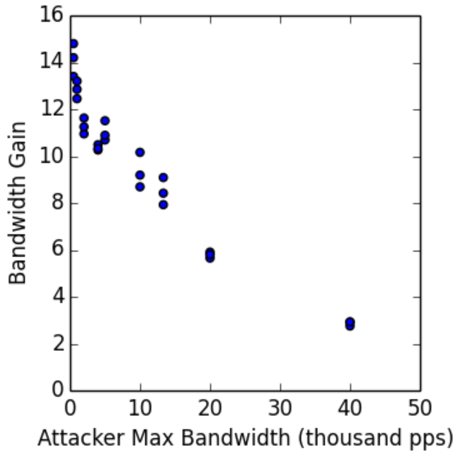


(c) Pulse Bandwidth

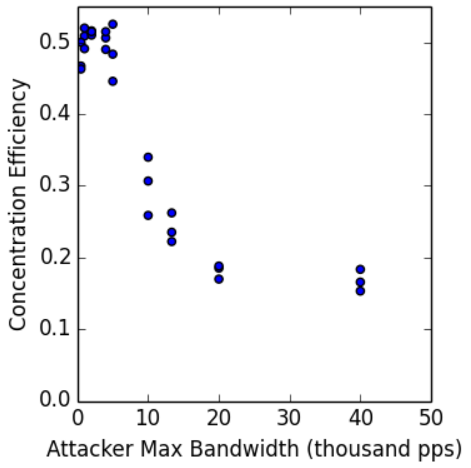
Figure 9: Lensing Metrics as a Function of *Throttled Bandwidth to Each Reflector* (with the AWS instance as victim—fixed dimensions: attacker bandwidth 10K pps, window size 20 ms)



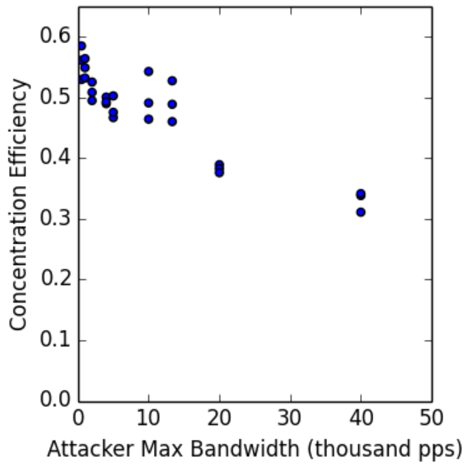
(a) Bandwidth Gain



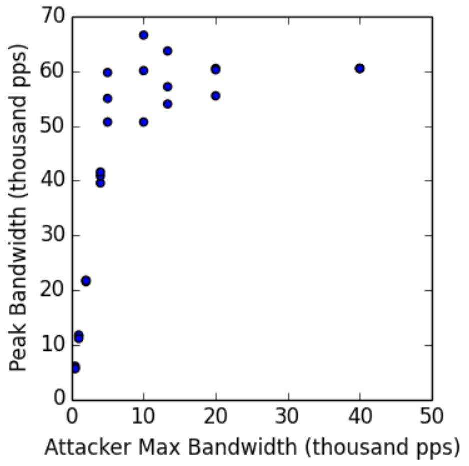
(a) Bandwidth Gain



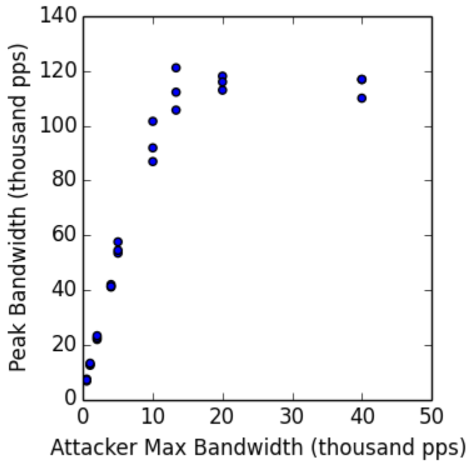
(b) Concentration Efficiency



(b) Concentration Efficiency



(c) Pulse Bandwidth



(c) Pulse Bandwidth

Figure 10: Lensing Metrics as a Function of the *Attacker's Maximum Bandwidth* (with the *Azure* Instance as victim—fixed dimensions: pulse window 20 ms, max bandwidth to reflector of 500 pps)

Figure 11: Lensing Metrics as a Function of the *Attacker's Maximum Bandwidth* (with the *AWS* instance as victim—fixed dimensions: pulse window 20 ms, max bandwidth to reflector of 500 pps)

because of high throttling of bandwidth to a constant size pool of reflectors (discussed in more detail section 6.2.2). The illuminating metric is that of concentration efficiency. We see little variation, except for at very high throttling (effectively sending a maximum of 1 or 2 packets to reflector), where there is high efficiency. With these high efficiencies, however, comes no bandwidth gain, meaning that there was no pulse created due to excessive throttling. Due to lack of variation in efficiency, we conclude that there is little to be gained by limiting the bandwidth to each reflector. That said, we did not investigate sending over 500 pps to a reflector for aforementioned reasons, and we are hesitant to extrapolate these results.

Figure 10 shows how the attack scales as a function of the attacker's bandwidth (we fix the pulse window to 20 ms and a maximum peak bandwidth of 500 pps to each reflector). The relatively constant efficiency at the beginning indicates that the attack scales well; the dropping bandwidth gain can be explained by the fact that we throttle bandwidth to each reflector while keeping the pool of available reflectors constant (as discussed in Section 6.2.2). However, we see all metrics perform poorly at higher bandwidths. Figure 10c offers a potential clue: it should scale linearly with the attacker's bandwidth, but instead we see it level off at about 50-60 thousand packets per second—meaning that the largest pulse we can create of duration 20 ms has bandwidth of 50-60K pps.

Now, there are two main explanations for this apparent pulse degradation at scale: either that (a) it is a feature of the attack scaling poorly, or (b) it is due to the attack working—increased jitter and queuing that would cause pulse flattening or packet loss would actually be expected in a successful attack.[7]

To determine the cause of the poor scaling of the Azure instance, we flooded it at a rate of 100K pps for a short duration. After three trials, we found the maximum download bandwidth for small DNS packets to be between 56.8K and 62.4K pps; thus, we conclude that the scaling issues are a sign of the attack's efficacy—namely that it saturated Azure's bottleneck resource in receiving these packets.

To further corroborate this attribution, we also tried the same tests on the AWS instance which we found can take more packets per second (Figure 11). Here the attack only starts to scale poorly at much higher peak bandwidths of about 110-120K pps—further evidence that the attack was not exhibiting poor scaling on the Azure instance. Unfortunately we were not able to generate more than a 100K pps[8] flood, so we were not able to directly determine with certainty what causes the poor scaling at 120K pps.

However, the difference in behavior between the Azure and AWS instances with regards to *unaccounted* for packets provides a hint in attributing the 120K pps leveling. We define unaccounted for packets as those that are sent by the attacker but whose reflection is never received at the victim. There are some subtleties here. Due to retransmits, almost all of the sent packets arrive eventually. So, we redefine "unaccounted" packets as those that do not arrive within 200 ms of the pulse window (which for the figures is only 20 ms) to avoid the potential problems with retransmitted packets.

Now, in Azure, it seems as if a router buffers the packets beyond the VM's quota when the burst is too high, as can be seen by the pulse spreading in Figure 12, but the AWS instance does not (compare to Figures 7b and 7c). Instead, the reflected packets never appear at the AWS end. Figures 13 and 14 quantify this difference. With the Azure instance as victim, we see a relatively constant proportion amount of unaccounted packets to those sent; thus, we see that the attack delivers most of the packets—even at high attacker bandwidths. However, at these higher bandwidths, the AWS victim receives much fewer of these packets, as shown in Figure 14. It appears as though AWS (most likely not the VM, but AWS routers) responds to an excessive download by dropping packets, instead of queuing them as does Azure. So, this discrepancy between Azure and AWS indicates that the attack is indeed effective (from the Azure results, we know almost all of the reflected packets arrive at the East Coast, but they are most likely just dropped by AWS routers before getting to our VM) and that the leveling-off at 120K pps is due to a limit on the end of AWS.

In short, the attack displays impressive numbers and scales well. As the peak pulse bandwidth nears the victim's maximum capacity, however, the attacker sees diminishing returns.

7 Extending the Attack

7.1 Attacks on Arbitrary End-hosts

We previously limited our attack to one on DNS nameservers. Since DNS typically operates over UDP with extremely short flows (if they can even be called such), the efficacy of a pulsing attack (which attacks TCP congestion control) is low. Clearly, an attacker would wish to target a more rewarding victim. However, to attack an arbitrary end-host (and not just a authoritative DNS

[7] Of course, we are assuming that measurement error is 0. To this end, we note that the packet capture tool we used, `tcpdump` reported no packet drops.

[8] The reader may wonder how we came up with the 110-120K pps poor scaling range while being unable to generate more than 100K pps. The answer is that the attacker's *sending* bandwidth could not exceed 100K pps, but at the *receiving* end of the attack, the packets concentrated into a higher 120K pps pulse

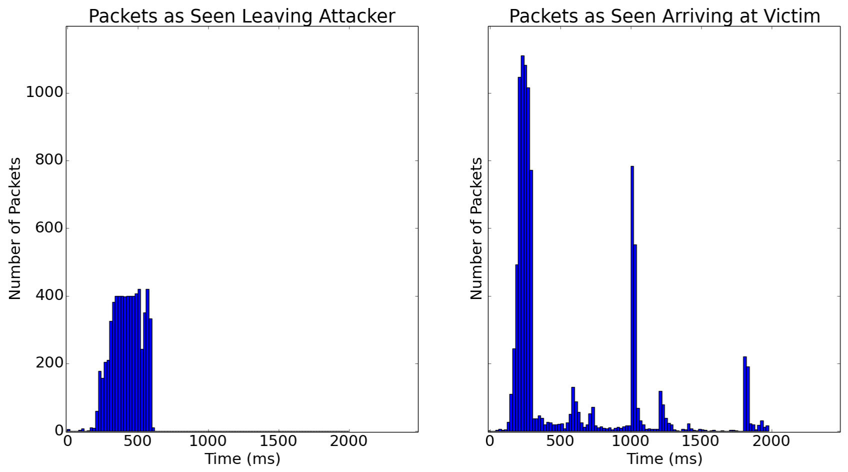


Figure 12: Illustration of Pulse Spreading at the Azure Victim (with dimensions of attacker bandwidth 20,000 pps, window size 20 ms)

nameserver), an attacker must somehow calculate attack path latencies to that host.

As a first step, we borrow a method from King [6] that uses something similar to DNS cache poisoning to calculate latencies between a DNS resolver and any other type of DNS server (not just an authoritative one). Say we want to calculate the latencies between a resolver NS_A and any type of DNS server (possibly a resolver) NS_B, as in Figure 15. Using the figure as an example, we make a DNS entry in *our own* authoritative DNS server—`mydomainname.com`—saying that NS_B `(10.0.0.0)` is authoritative for `10-0-0-0.mydomainname.com`. Then, if we issue queries to NS_A for subdomains of `10-0-0-0.mydomainname.com`, NS_A will have cached the NS record that NS_B is authoritative for `10-0-0-0.mydomainname.com` and it will query NS_B. NS_B will reply with an error, but the chain of queries will reveal to us the attack path RTT.

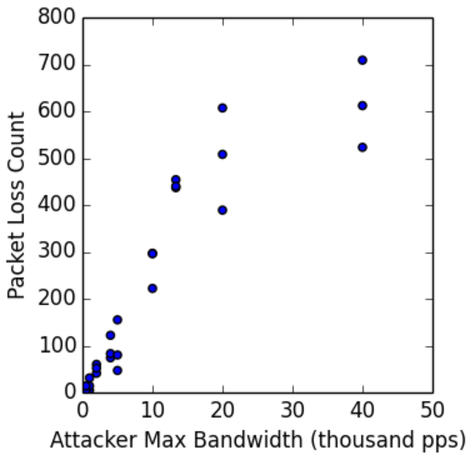
Now, we propose our own addition to King's cache-poisoning trick to extend it to arbitrary end hosts. We replace NS_B, which in the above example is any type of DNS server, with B any arbitrary server (not necessarily DNS), but keep the rest the same. Now, when B receives a DNS query from NS_A, it most likely won't have a service running on port 53 (DNS). According to RFC 1122 [2], B "SHOULD" respond with an ICMP Port Unreachable *and* that in response to such a message, the UDP layer of NS_A "MUST" pass an error up to the application layer. If NS_A's DNS implementation responds to the error passed up, and immediately responds to us, we again can calculate the attack path RTT.

There are two caveats to the above method. First, B may not even receive a packet on port 53 due to a firewall, or it may be explicitly configured not to respond. Second, the resolver implementation must respond to ICMP error messages passed up and deal with them appropriately. Our server, using the default configuration of BIND9 on Ubuntu Linux, does in fact do both: it issues an ICMP error when a service is not running on port 53; as a resolver, BIND9 will immediately respond back to the client when obtaining an ICMP. Further, we have tested this method to build schedules and create pulses and have found that it does indeed work with many resolvers; some resolvers, however, do not react to the ICMP and instead timeout, leaving us with no latency data. In short, we expect this addition to King to allow us to better estimate path latencies to many more types of hosts than just DNS servers.

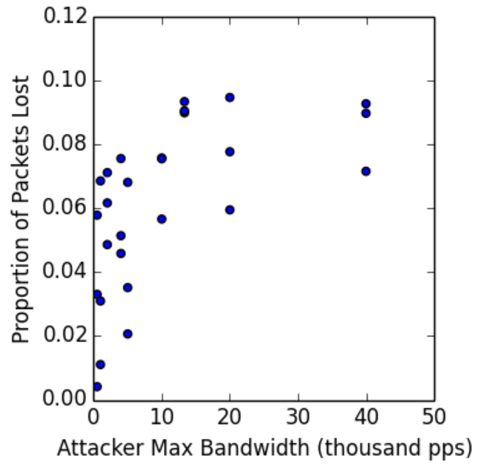
Lastly, the simplest method is to attempt to find a DNS server co-located with the actual victim. As noted in [6], this is relatively common, but may introduce error in latency measurements of the attack path. However, if this error is the same for each attack path (as would be expected for a DNS server in the same network as the victim), then this error will have no effect on the actual pulses. For example, if every measurement has error of +20 ms, then each reflected packet will arrive at the victim at 20 ms before it is expected, but all packets will still arrive at the same time.

7.2 Amplification

A natural extension to concentrating a flood in time is to make the flood larger via amplification. Indeed, a common use for reflectors is amplification. Both amplification and lensing can compatibly leverage open DNS resolvers as reflectors—in fact resolvers are already used in amplification attacks. The idea would be to take at-

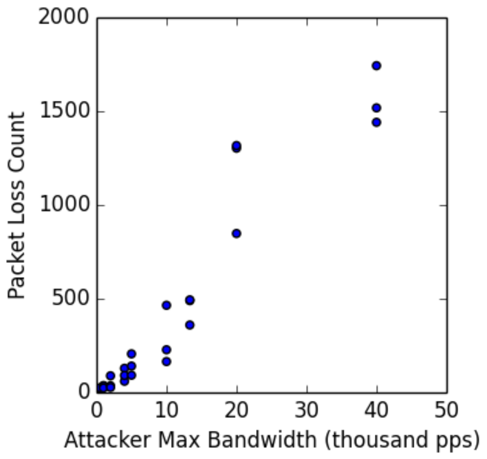


(a) Absolute Number of Unaccounted Packets

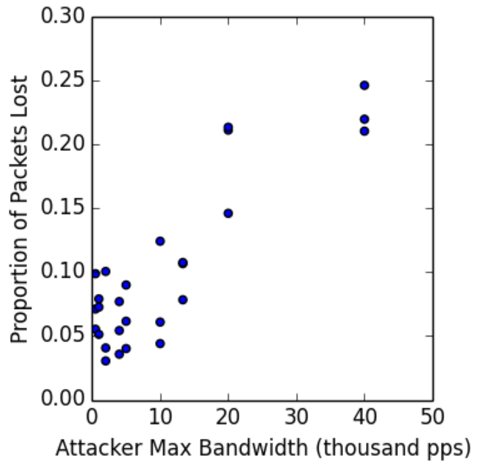


(b) Proportion of Unaccounted Packets

Figure 13: Unaccounted Packets as a Function of Attacker's Maximum Bandwidth (with the *Azure* instance as victim)



(a) Absolute Number of Unaccounted Packets



(b) Proportion of Unaccounted Packets

Figure 14: Unaccounted Packets as a Function of Attacker's Maximum Bandwidth (with the *AWS* instance as victim)

tack path latency measurements through reflectors using the methods we borrow from King and during the actual attack use these same reflectors for amplification.

Note that the form of lensing we have explored does not use source address spoofing to enable reflection; instead it relies on recursive queries. However, DNS-based amplification does require spoofing. While spoofing should work just fine with lensing, we decided against it for our experimentation for simplicity and also to avoid confusing analysts potentially investigating our traffic.

In such a scenario, the attacker would gain the best of both worlds. For example, an amplification factor of 15 and lensing bandwidth gain of 10 could, at its worst, allow attackers to create pulses at 150 times their uplink bandwidths!

Forwarders (discussed in Section 4.1) introduce a potential caveat to estimating attack path latencies in amplification attacks. If the resolver the attacker contacts is indeed a forwarder, then during the attack path latency measurements, there are two intermediary hops between attacker and victim instead of just one. This was not a problem for our attack, since attack traffic followed the same path as latency measurements (attacker $\Rightarrow$ forwarder $\Rightarrow$ resolver $\Rightarrow$ victim). However, in an amplification attack, it would be expected that the attacker would place a large query response in a cache. If it is cached in the forwarder, then the path of the actual attack traffic will be one hop shorter (attacker $\Rightarrow$ forwarder $\Rightarrow$ victim).

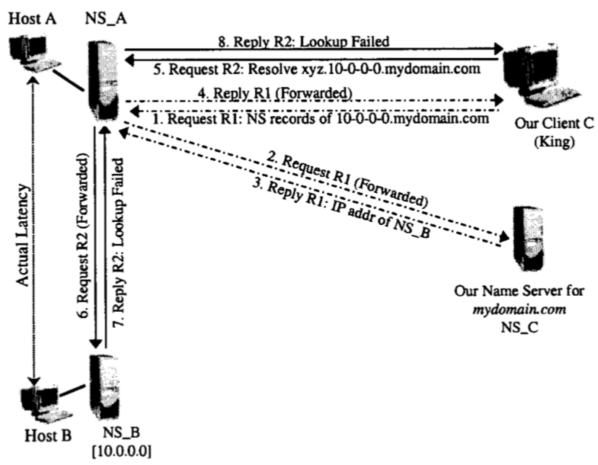

Host A
NS_A
8. Reply R2: Lookup Failed
5. Request R2: Resolve xyz.10-0-0-0.mydomain.com
4. Reply R1 (Forwarded)
1. Request R1: NS records of 10-0-0-0.mydomain.com
Our Client C
(King)
2. Request R1 (Forwarded)
3. Reply R1: IP addr of NS_B
Actual Latency
6. Request R2 (Forwarded)
7. Reply R2: Lookup Failed
Our Name Server for
mydomain.com
NS_C
Host B
NS_B
[10.0.0.0]

Figure 15: The sequence of queries to perform the King cache poisoning trick (reprinted with approval from [6]). The dotted lines represent queries to warm up caching the NS record. The thick lines allow measurement of the attack path RTT.

Turbo King [11], building off of King, noticed that King overlooked forwarders and added the ability to detect forwarders, by essentially positioning themselves at both ends of the DNS query (in our example, as attacker and victim). An attacker who sets up a personal DNS server can do the same: identify potential forwarders, measure the error they introduce, and weed them out as needed.

Thus, as we have shown the efficacy of lensing on its own, we stress that its combination with amplification may make for a particularly potent threat.

7.3 Distributed Attack

A natural extension would be to use many geographically distributed machines and try to "add" their pulses together at the victim. This extension would require relatively accurate time synchronization between the attack seeders, depending on the desired pulse duration. The simplest way would be to use a time synchronizing protocol (such as NTP). Preliminary experimentation has shown that NTP yields a high enough imprecision that two attackers cannot reliably create larger pulses of window size 20 ms, but they can at window sizes of 40 ms.

7.4 Increasing the Bandwidth Gain

7.4.1 Increasing the attack-path latency

With longer attack path latencies, an attacker has a longer period to send and thus funnel bandwidth. Currently, the longest attack path latencies we find are around 800 ms. We propose a way to extend the time a query takes, but keep the time predictable.

One idea is to use IP spoofing as the reflection mechanism. For each resolver, we ask it to make a query that will take a long time (for example if we query for a domain far from the resolver). When the resolver finally obtains a response, it will respond to the victim a considerable amount of time later.[9] As a slight variation, the attacker might make a query to a DNS server that does not respond, causing the resolver to timeout and send back a negative response after the timeout. Note that the timeout can be measured in advanced with essentially 100% precision (since timeouts are usually round numbers).

Thankfully, a traditional DNS-based amplification attack is not compatible with any of these extensions because the amplified DNS payload is kept cached in the resolver. So when the resolver is queried, it immediately responds with the cached answer giving no chance to increase the attack path latency.

7.4.2 Retransmits

As we saw in Figures 7b and 7c, resolver retransmits can create secondary pulses of their own. The retransmit timeouts that were the most common were 800 milliseconds and 2 seconds.

An attacker can predetermine which set of resolvers have what timeouts. Then, this attacker can arrange to send pulses at a period of these timeouts, so that new pulses coincide with pulses generated from retransmits. The attacker would then essentially be superimposing pulses—allowing an even larger bandwidth gain.

Again, this attack is not compatible with IP spoofing (and thus amplification). With spoofing, traffic reflected off of the resolver consists of query responses (instead of queries), and resolvers will not retransmit responses (they only retransmit queries for which they do not get an answer for).

8 Defenses

8.1 Detecting and Thwarting Reconnaissance

The method by which we conducted reconnaissance (obtained attack path latency measurements) was extremely noisy since the victim was notified on every measurement with a query for a bogus subdomain. Attackers,

[9]There are probably many misconfigured DNS entries to help an attacker here. Also, an attacker can intentionally misconfigure a personal DNS server to this end, for example, adding an NS entry to server that will not respond.

however may be able to hide their presence by making queries for legitimate subdomains.[10]

Perhaps the victim can *thwart* reconnaissance by poisoning attack path RTT measurements. The victim can, for example, introduce artificial jitter. The most obvious approach, just adding a random delay to each request will only slow down an attacker—by taking more measurements an attacker can cut through the noise and determine actual latencies. Instead, the victim might introduce an amount of jitter as a function of the resolver's address. Ideally this function would be keyed and cryptographically secure in the sense that knowing one jitter would not reveal information of another.

However, none of this analysis considers that an attacker may be able to take measurements to a nearby server *not under the victim's control*. In this case, the victim will not be able to detect reconnaissance at all. Without detection, the victim has little chance of thwarting reconnaissance.

8.2 Detecting and Preventing Attacks

[8] suggests two methods to mitigate bursty attacks targeting TCP's RTO mechanism: RED and RTO randomization. The former method proves largely ineffective, while the latter involves a trade-off with throughput in the absence of an attack. [12] discusses a method to simply detect the presence of a pulsing DoS attack based on the observation in high variability traffic to the victim (characteristic of a pulse) and subsequent decline in ACK traffic back. However, our focus here is on how one might defend against the *lensing* side of the attack.

Again, our defense revolves around introducing arbitrary jitter. A possible mechanism would be for routers to somehow add jitter during an attack. While it is unclear how they might coordinate such a mechanism without introducing collateral damage, we explore how it might play out in Figure 16. We add a uniform, random amount of jitter after a sending schedule is produced, which is essentially the same effect routers adding jitter would have. The graph indicates that cutting a relatively small window pulse of 20 ms in half would require adding jitter of 60 ms[11] to the attack path.

In short, smart attackers can completely hide their reconnaissance from the victim. Mitigating an actual lensing attack would require somehow changing attack path latencies during an attack; it is unclear how to create a mechanism to do so without harming legitimate traffic, and even then, a significant amount of jitter needs to be added. We unfortunately conclude that mitigating this vector is likely impractical. Possibly a more fruitful angle would be to tackle the pulsing side of the attack (i.e., improving TCP congestion control robustness) rather than the lensing side.

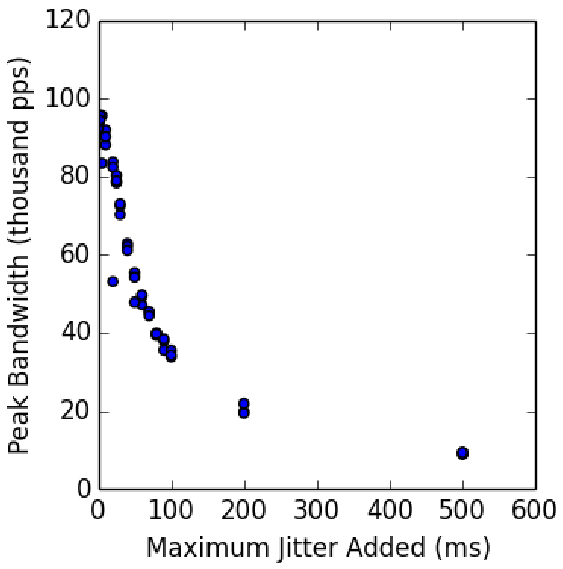


Figure 16: Pulse Degradation as with the Addition of Artificial Jitter (parameters of: pulse window 20 ms, sending rate 10K pps)

9 Future Work

9.1 Determining Impact on TCP Flows

Our work solely focused on concentrating a flood temporally and the properties of such lensing. We have done some preliminary experimentation with its effects on TCP congestion control—simulating a relatively low-bandwidth attacker, we have been able to bring a higher bandwidth TCP flow to its knees. However, we have not performed methodical measurements.

While, previous work has already demonstrated the damaging effects of pulsing, the pulses created here differ in two important ways. First, they are not square wave pulses, but are markedly more normally distributed. Second, they are generated by multiple sources, meaning routers close to the victim will deal with not just many arrivals, but on different ports. This point may also make it more difficult for the victim to filter malicious traffic.

[10]It is true that the attacker will need to query distinct subdomains for each reflector to avoid hitting the reflector's cache. However, we only used 5-10 queries to obtain latency measurements, so the attacker would just need to find a few unique domain names.

[11]By this we mean adding a uniformly distributed amount of jitter in the range 0-60 ms

9.2 Experimentally Validating Extensions

In Section 7 we discussed several ways to improve the basic lensing attack we have developed—chief among them being combining amplification with lensing. While theoretically sound, it would be ideal to validate them in practice and determine just how effective they are.

10 Conclusion

We introduce the idea of temporal lensing, which lends itself quite naturally to pulsing DoS attacks. Using DNS recursion to both estimate attack path latencies and to create pulses from relatively low-bandwidth floods, we further demonstrate its practicality and explore some of its properties. We find that lensing allows an attacker to concentrate the bandwidth of a flood by an order of magnitude. Given these results, lensing's compatibility with amplification, and our inability to find suitable defenses, we stress that the potential for abuse is high.

11 Acknowledgements

We are grateful to Mark Allman and Nick Weaver for their helpful advice.

References

[1] Public DNS Server List, May 2014. available at `http://public-dns.tk/`.

[2] Braden, R. RFC 1122: Requirements for Internet Hosts. *Request for Comments* (1989), 356–363.

[3] Dabek, F., Cox, R., Kaashoek, F., and Morris, R. Vivaldi: A Decentralized Network Coordinate System. In *ACM SIGCOMM CCR* (2004), vol. 34, ACM, pp. 15–26.

[4] Francis, P., Jamin, S., Jin, C., Jin, Y., Raz, D., Shavitt, Y., and Zhang, L. IDMaps: A Global Internet Host Distance Estimation Service. *Networking, IEEE/ACM Transactions on 9*, 5 (2001), 525–540.

[5] Guirguis, M., Bestavros, A., Matta, I., and Zhang, Y. Reduction of Quality (RoQ) Attacks on Internet End-systems. In *INFOCOM. 24th Annual Joint Conference of the IEEE Computer and Communications Societies* (2005), vol. 2, IEEE, pp. 1362–1372.

[6] Gummadi, K. P., Saroiu, S., and Gribble, S. D. King: Estimating Latency Between Arbitrary Internet End Hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment* (2002), ACM, pp. 5–18.

[7] Kogler, T. M. Single Gun, Multiple Round, Time-on-Target Capability for Advanced Towed Cannon Artillery. Tech. rep., DTIC Document, 1995.

[8] Kuzmanovic, A., and Knightly, E. W. Low-rate TCP-targeted Denial of Service attacks: The Shrew vs. the Mice and Elephants. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2003), ACM, pp. 75–86.

[9] Ledlie, J., Gardner, P., and Seltzer, M. I. Network Coordinates in the Wild. In *NSDI* (2007), vol. 7, pp. 299–311.

[10] Ledlie, J., Pietzuch, P., and Seltzer, M. Stable and Accurate Network Coordinates. In *Distributed Computing Systems. ICDCS. 26th IEEE International Conference* (2006), IEEE, pp. 74–74.

[11] Leonard, D., and Loguinov, D. Turbo King: Framework for Large-scale Internet Delay Measurements. In *INFOCOM. The 27th Conference on Computer Communications. IEEE* (2008), IEEE.

[12] Luo, X., and Chang, R. K. On a New Class of Pulsing Denial-of-Service Attacks and the Defense. In *NDSS* (2005).

[13] Moore, D., Shannon, C., Brown, D. J., Voelker, G. M., and Savage, S. Inferring Internet Denial-of-Service Activity. *Transactions on Computer Systems 24*, 2 (2006), 115–139.

[14] Ng, T. E., and Zhang, H. Predicting Internet Network Distance with Coordinates-Based Approaches. In *INFOCOM. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings* (2002), vol. 1, IEEE, pp. 170–179.

[15] Paxson, V. An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks. *ACM SIGCOMM CCR 31*, 3 (2001), 38–47.

[16] Schomp, K., Callahan, T., Rabinovich, M., and Allman, M. On Measuring the Client-Side DNS Infrastructure. In *ACM SIGCOMM/USENIX Internet Measurement Conference* (Oct. 2013).

[17] Sharma, P., Xu, Z., Banerjee, S., and Lee, S.-J. Estimating Network Proximity and Latency. *ACM SIGCOMM CCR 36*, 3 (2006), 39–50.

[18] Wong, B., Slivkins, A., and Sirer, E. G. Meridian: A Lightweight Network Location Service without Virtual Coordinates. *ACM SIGCOMM CCR 35*, 4 (2005), 85–96.

12 Appendix: Proof of Send Schedule Optimality

For each time $t \in T$ that we consider sending and for each reflector $r \in R$, we have $\Pr(t,r)$, the probability that if we send at reflector r at time t the reflected packet will land in the desired window—note that Section 4 gives us estimates for this probability.

Say we have chosen a schedule for which at each time t we send to reflector r_t. Let X be the random variable denoting how many packets arrive in the window. $X = \sum_{t \in T} X_t$ where

$$X_t = \begin{cases} 1 & \text{if packet sent at } t \text{ lands in window} \\ 0 & \text{otherwise} \end{cases}$$

So, $E(X_t) = \Pr(t, r_t)$. Then, due to linearity of expectation,

$$E(X) = E(\sum_{t \in T} X_t) = \sum_{t \in T} E(X_t) = \sum_{t \in T} \Pr(t, r_t)$$

.

Now, we claim that any schedule that optimizes $E(X)$ must have the condition that for each t, we send to the reflector with highest $\Pr(t,r)$ over $r \in R$. To see this, assume for the sake of contradiction that there is some optimal schedule S such that at time t^* we do not send to the reflector r^* that yields the highest probability, instead we send to r^{**}. Since r^{**} is not the best, $\Pr(t^*, r^*) > \Pr(t^*, r^{**})$. Consider S' which is the same schedule as S except that at t^*, it sends to r^*. Let the expected number of packets landing in the window of S and S' be $E(X)$ and $E(X')$ respectively; then the difference between the expectation of the schedules is

$$\begin{aligned}
E(X') - E(X) &= \sum_{t \in T} E(X'_t) - \sum_{t \in T} E(X_t) \\
&= (\sum_{t \in T \wedge t \neq t^*} E(X'_t) + \Pr(t^*, r^*)) \\
&\quad - (\sum_{t \in T \wedge t \neq t^*} E(X_t) + \Pr(t^*, r^{**})) \\
&= (\sum_{t \in T \wedge t \neq t^*} E(X'_t) - \sum_{t \in T \wedge t \neq t^*} E(X_t)) \\
&\quad + (\Pr(t^*, r^*) - \Pr(t^*, r^{**})) \\
&= \Pr(t^*, r^*) - \Pr(t^*, r^{**})
\end{aligned} \tag{1}$$

But, we already established that this last term is greater than 0. Thus, $E(X_{S'}) - E(X_S) > 0$ and S' is better than S, which contradicts our assumption that S is optimal.