ipt_geofence - Performance Evaluation

Francesco Lorenzoni, Yuri Caprini, Salvatore Guastella

June 16, 2022

Contents

Abstract

ipt_geofence is a tool developed during the 21/22 *Network Management* UniPi course , that allows you to protect a host or a network by blocking incoming and outgoing connections according to some of their properties deduced from the analysis of their related packets.
Currently *ipt_geofence* is able to block connections from/to unwanted countries (*geofencing*), unwanted hosts (*blacklisting*) or hosts connecting on user-defined port ranges (*honeypot*).
Under normal network conditions *ipt_geofence* performances are extremely good: even if packets inspection could result costly, *ipt_geofence* needs to analyze only one packet[1] per connection to decide whether to block it or not, making the overall impact on the network negligible.
But what happens when *ipt_geofence* has to handle a lot of connections per second?
This work aims to evaluate *ipt_geofence* performances and behaviour in this specific worst-case scenario using commodity hardware as test bed.

[1] *ipt_geofence* gets a copy of the whole packet, but analyzes IP/Transport header only

1 Architectural overview

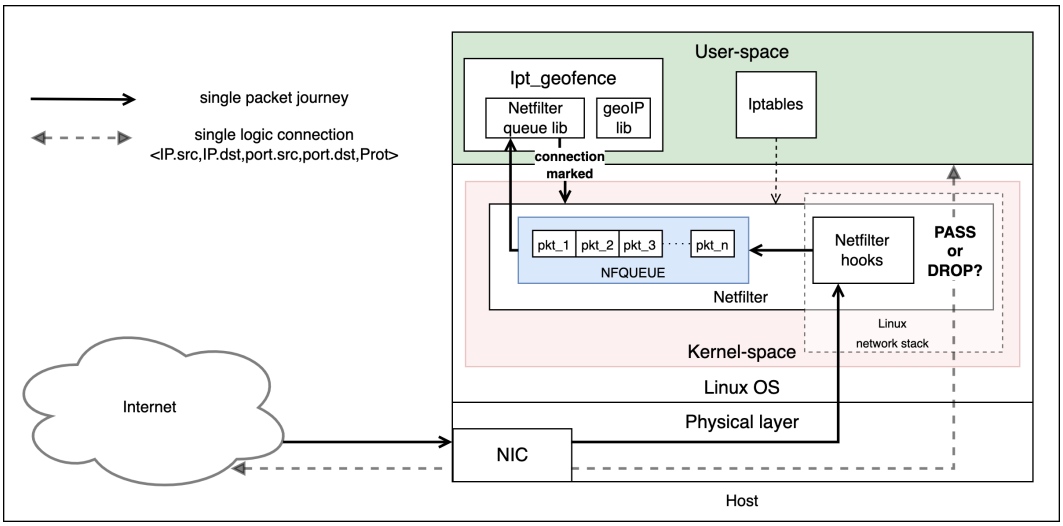

Figure 1: How *ipt_geofence* works under the hood.

Before conducting any experiment, we have to deeply understand how *ipt_geofence* works under the hood, in order to define which software behaviors need investigation, which measurements should be made and how to make them.
We know that a packet sent from or received by a host has to go through several stages of the Linux network stack. *netfilter* is an open source project which provides a one to one mapping of these stages with so called *netfilter* hooks. Before *ipt_geofence* execution, *iptables*, another popular firewalling software that allows you to define rulesets, is used to configure these hooks to perform some actions on the packets that arrive at certain stages of the network stack.

When a packet comes from an interface or user space it reaches the PREROUTING and OUTPUT phase of the network stack respectively. In these stages *netfilter* hooks are set up to queue these packets into a fixed-size NFQUEUE which resides in kernel space.

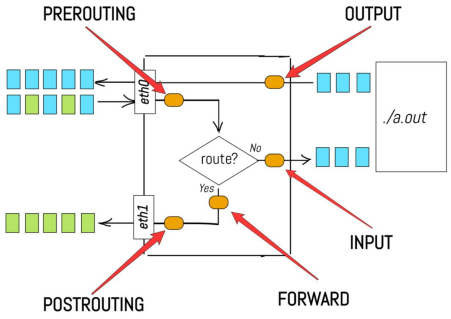

Figure 2: netfilter hooks diagram

ipt_geofence performs an infinite loop, popping out packets from this queue as fast as it can, analyzing them and marking the related connection (identified by a quintuple composed

by source IP, destination IP, source port, destination port and protocol id) with a value[2]. All packets belonging to a *marked* connection are **automatically** dropped or passed depending on the mark value. This means that they are not even queued into NFQUEUE and consequently *ipt_geofence* does not have to analyze them.
This is great: only **one** packet per connection needs to be analyzed and even if can be costly, we can guess that the overall performance degradation introduced by *ipt_geofence* on a network under ordinary operating conditions is reduced to minimum.

2 Worst-case scenario

What happens when *ipt_geofence* sees a large number of *unmarked* connections per second? A large number of packets will eventually fill the queue and when this happens, following packets that would to be pushed, are dropped by the kernel by default instead. This means that as long as the queue is full no other unmarked connections can reach or leave the host on which *ipt_geofence* is running. It can be worse: if the host is placed at the network **edge** acting as single router, the whole network will become isolated. Depending on the use case, this can be either wanted or unwanted behavior for the user, but for us it would be interesting to answer to this question:

How many unmarked connections per second can ipt_geofence *handle without dropping packets?*

We know from queue theory that when the push-rate (ν_{push}) of the elements pushed into a fixed-length queue is greater than the pop-rate (ν_{pop}), — *assuming they do* ***not*** *vary over time* — after a certain interval of time that queue will become **full** (discarding subsequent pushed packets in our case study) and it will remain in that state as long as the initial conditions persist. On the contrary, if the ν_{push} is less than or equal to the ν_{pop}, there is no queuing time for elements pushed to the queue.
A queue acts like a *"sponge"*: when $\nu_{push} > \nu_{pop}$ it can absorb the elements for a certain period of time, but when this one has elapsed it will become full and begin to drip.
When $\nu_{push} \leq \nu_{pop}$ the queue is basically useless as the entity that pops the element from the queue is always ready to take next one as soon as it is pushed to the queue. In our case study "the entity" is obviously *ipt_geofence*, the queue is NFQUEUE, ν_{push} is equal to the rate of packets belonging to unmarked connections that are pushed to NFQUEUE, and $\nu_{pop} = \frac{1}{T_e}$ where T_e is the *ipt_geofence* average per packet elaboration time.
This means that once we have estimated T_e, we can also answer to our initial question, calculating the maximum ν_{push} at which *ipt_geofence* does not (eventually) lose packets. This is when:

$$\nu_{push} = \nu_{pop} = \frac{1}{T_e}$$

Due to the queue presence and environment limits, calculating T_e will be not a straightforward task. We will have to make some (reasonable) assumptions and think about the correct way to measure it. All of this will be discussed in the next section.

[2] It's either a PASS or DROP mark, but the value itself is configurable

3 Measuring T_e

Our approach to T_e calculation starts from setting up a simulation environment composed by two machines that we will call $host_A$ and $host_B$. A single network interface (NIC_A) on $host_A$ is physically connected to a single (NIC_B) on $host_B$, creating a private network of two nodes, isolated from the rest of the world. On $host_B$ resides a running instance of *ipt_geofence*, while on $host_A$ resides a running instance of *wireshark/tcpdump* capturing packets passing through NIC_A.
Using *tcpreplay*, $host_A$ sends to $host_B$ a sample of n packets each belonging to a different connection, at a configurable send rate. $host_B$ is configured to route all the packets received on NIC_B to $host_A$. When the simulation ends $host_A$ has captured all packets sent to and received from $host_B$ and has stored them in a single `.pcap` file.

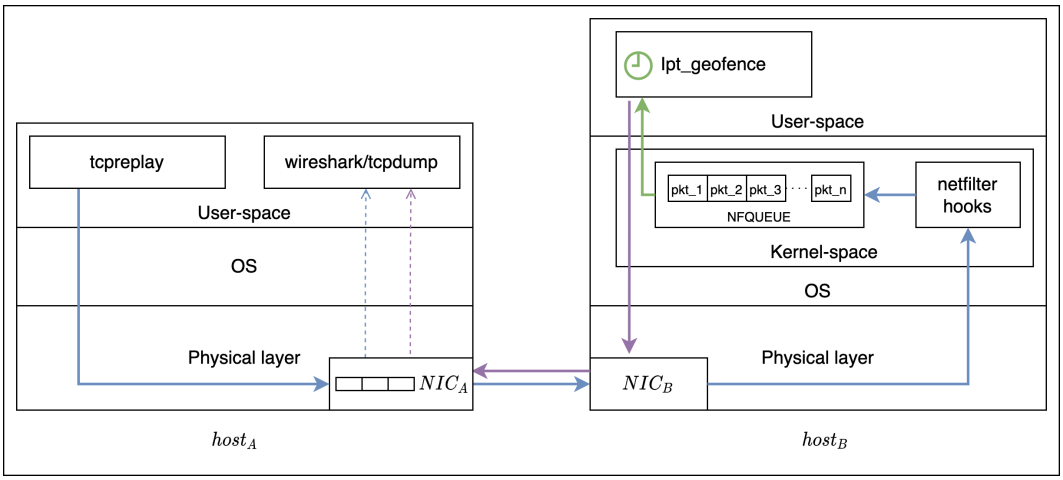

Figure 2: an example of simulation in our test environment.

In order to calculate T_e from this packet capture we need to do some considerations. First of all, let's define:

- T_{A2Q} as the average time interval a packet takes to go from NIC_A to inside NFQUEUE
- T_{G2A} as the average time interval a newly analyzed packet takes to go from *ipt_geofence* to NIC_A

Under these assumptions we can also tell that packets sent rate is equal to ν_{push}. Since packets sent rate is configurable, we can choose ν_{push} arbitrarily in our simulations. Now let's take a look at Fig. 3 showing how the round trip time of the packets sent varies on the choice of ν_{push}:

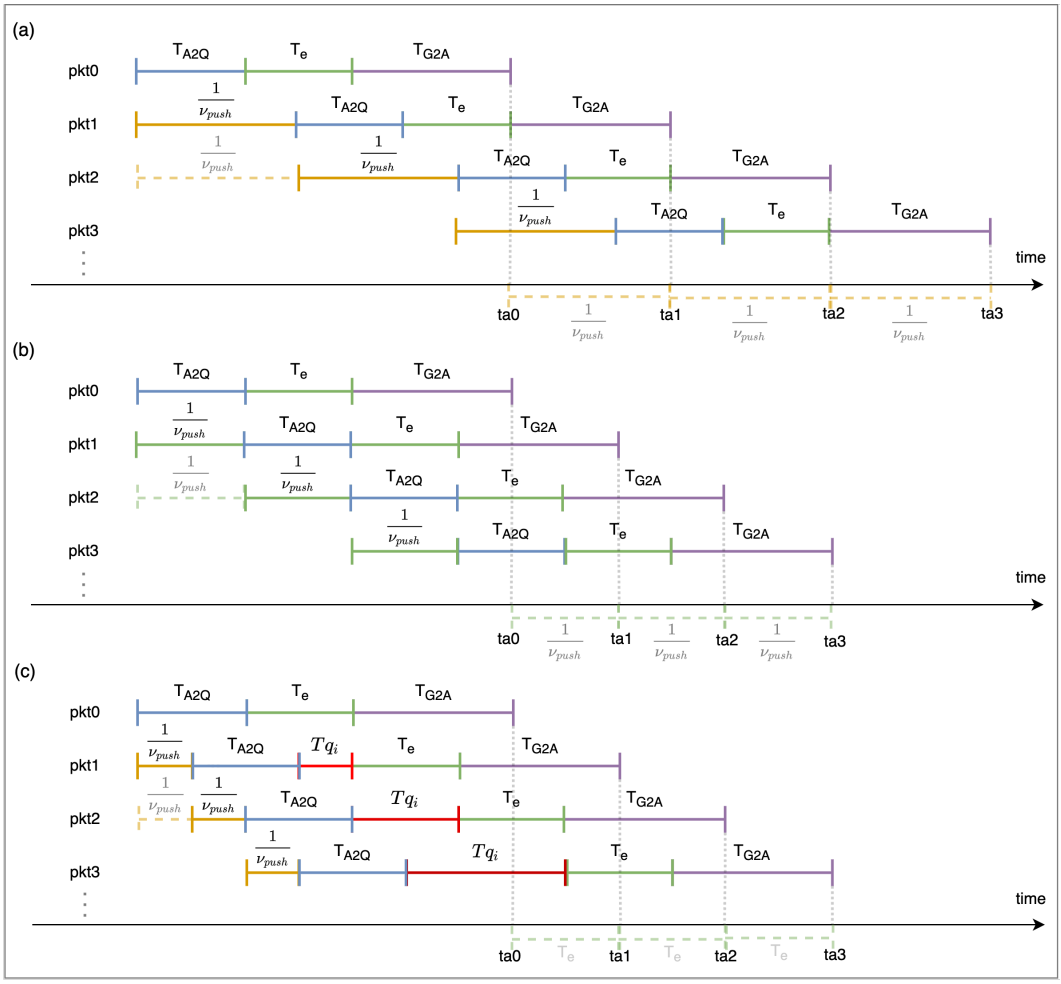

Figure 3: how the round trip time of the packets sent varies on the choice of ν_{push}

(a) $\frac{1}{\nu_{push}} > Te \Rightarrow \nu_{push} < \nu_{pop}$ and the queuing time for each sent packet i is $Tq_i = 0$

(b) $\frac{1}{\nu_{push}} = T_e \Rightarrow \nu_{push} = \nu_{pop}$ and $Tq_i = 0$

(c) $\frac{1}{\nu_{push}} < T_e \Rightarrow \nu_{push} > \nu_{pop}$ and $Tq_i > 0$

We can also make another useful observation. Defining ta_i as the instant of arrival of a packet i to $host_A$ after being sent to $host_B$, we note that:

(a) $ta_i - ta_{i-1} = \frac{1}{\nu_{push}} > T_e$

(b) $ta_i - ta_{i-1} = \frac{1}{\nu_{push}} = T_e$

(c) $ta_i - ta_{i-1} = T_e < \frac{1}{\nu_{push}}$

3.1 Measurement Methods

Given the above mentioned considerations we can finally determine two methods to estimate T_e:

1. **Method 1**: We can choose a very low ν_{push} aiming to keep $\frac{1}{\nu_{push}} \geq T_e$, making $Tq_i = 0$ throughout the simulation (see our first consideration). We can achieve this with certainty by sending a packet only when the previous one has returned to $host_A$. At this point we can run two simulations (with a sample of n packets each), one with $host_B$ running an instance of *ipt_geofence* and one without. Once we have the resulting pcap files ($pcap_1$ and $pcap_2$) we perform the following procedure: for each packet i in $pcap_1$ and $pcap_2$ we calculate the round trip time ($RTT1_i$ for packet i in $pcap_1$ and $RTT2_i$ for packet i in $pcap_2$) and then we can obtain T_e as follows:

   $$\mathrm{T}_e = \sum_{i=1}^{n} \frac{|RTT1_i - RTT2_i|}{n}$$

   Note that this method can't be used to calculate T_e if $\frac{1}{\nu_{push}} < T_e$ because in that case we would have $RTT1_i - RTT2_i = Te + Tq_i$ with $Tq_i > 0$.

2. **Method 2**: We can choose a very high ν_{push} aiming to keep $\frac{1}{\nu_{push}} < T_e$. In this case Tq_i will continue to grow over the time. We run a single (standard) simulation with a sample of n packets and keeping in mind previous considerations (see our second consideration) we can obtain average T_e from:

   $$\mathrm{T}_e = \sum_{i=1}^{n} \frac{|ta_i - ta_{i-1}|}{n}$$

   Note that if we use this method when $\frac{1}{\nu_{push}} > Te$ we obtain:

   $$\frac{1}{\nu_{push}} = \sum_{i=1}^{n} \frac{|ta_i - ta_{i-1}|}{n} > T_e$$

   while if we use this method when $\frac{1}{\nu_{push}} = T_e$ we obtain:

   $$\frac{1}{\nu_{push}} = \sum_{i=1}^{n} \frac{|ta_i - ta_{i-1}|}{n} = T_e$$

So now we have two different methods for calculating T_e, each with its pitfalls. In the next section we will conduct our tests and we will have to choose which method to use and say why.

4 Test bed and results

We have conducted our simulations using a machine equipped with *intel i7-8750H/16GB RAM* as $host_B$. For each simulation, 10^5 packets are sent from $host_A$ using *tcpreplay*. These packets have been forged (and stored in a `.pcap` file) using *scapy* so that each packet belongs to a different connection.
To calculate T_e we decided to use method 2. We'll discuss later why.

We have performed a first simulation by choosing the maximum packets sending rate (ν_{push}) possible with our setup, aiming to keep $\nu_{push} > \nu_{pop}$ and falling back to case (c) of Fig. 3 Let's see the result:

Table 1:

ν_{push} [pps]	T_e [s]	$\sigma^2_{T_e}$ [s^2]	$\nu_{pop} = 1/T_e$ [pps]
267597	0.00001596496	1.500452638515512e-10	62637

The result reflects what we expected: $\nu push > (\nu_{pop} = \frac{1}{T_e})$ or in other terms $\frac{1}{\nu_{push}} < T_e$. Under these conditions we know that the value returned by method 2 is truly T_e. This means that $\nu_{push} \approx 60000$ $pps (\approx \nu_{pop} = \frac{1}{T_e})$ is the maximum value of **unmarked** connections that *ipt_geofence* can handle without eventually lose packets on this machine. We have reached our goal!

As a further confirmation we ran more simulations by setting different values of ν_{push}. We expected that using method 2:

(a) for $\nu_{push} < 60000$ pps we should fall back to case (a) of Fig.3 and in that case the value returned by method 2 is not T_e (although for simplicity in the results table we will leave it indicated like this), but $\frac{1}{\nu_{push}}$. This means that we should see from the results table $\nu_{push} \approx \nu_{pop} < 60000$ pps.

(b) for $\nu_{push} \approx 60000$ pps we should fall back to case (b) of Fig.3 and the value returned by method 2 is $\frac{1}{\nu_{push}} = T_e$. From the results table we should see $\nu_{push} = \nu_{pop} \approx 60000$ pps.

(c) for $\nu_{push} > 60000$ pps we should fall back to case (c) of Fig.3 and the value returned by method 2 is T_e. From results table we should see $\nu_{push} > \nu_{pop} \approx 60000$ pps.

ν_{push} [pps]	T_e [s]	$\sigma^2_{T_e}$ [s^2]	$\nu_{pop} = 1/T_e$ [pps]
30000	0.00003332439	3.1767559273379277e-10	30008
40000	0.00002499035	6.499254909564009e-10	40015
50000	0.00001999335	4.0704587889583243e-10	50016
60001	0.00001712912	2.797280112384336e-10	**58380**
70002	0.00001785563	2.4869696603783193e-10	**56004**
80003	0.00001792802	2.5421742691488236e-10	**55778**
90003	0.00001288364	7.752105567923821e-11	77617
100004	0.00001234847	7.645837839578344e-11	80981

Simulations with $\nu_{push} = 90|100k$ display odd behaviour, even though the overall results seem to match our expectations, in fact $\nu_{push} = 60|70|80k \Rightarrow 60000$ pps, which is what we expected from the values above.

5 Observations

5.1 Older hardware

We've replicated some of the tests described above on a much less newer PC (`i5-4210U`, launched in Q2'14).
We haven't displayed all the data gathered on this machine to avoid redundancy (since the considerations made are almost identical), but it might be of interest to note how T_e (method 2) and $max\{\nu_{push}\}$[3] change along with the hardware:

$$T_e = 0.00008903 \qquad max\{\nu_{push}\} \approx 10k$$

As you can see, both of the above values differ from newer hardware of about a x6 factor.

5.2 About method 1

We've preferred method 2 over method 1 due to some considerations related to this latter one.
First of all, due to our limited resources, we could not perform testing using low ν_{push} with a proper sample size, since doing so takes a lot of time. On long lasting tests, we've noticed that the captured data was *dirty*: *RTT* didn't look stable and some local traffic used to occur. We've tried using a *BPF filter* when capturing to eliminate undesired traffic, but a clean capture, without this tricks, was preferable.
However, we gathered some data, using a little sample of 10^3 packets, taking into account that it could have led to inaccurate results. Comparing T_{e_1} —T_e resulting from applying method 1 to these data— and T_{e_2} —a previous calculation of T_e using method 2 on data gathered with other simulations on the same machine— we should have $T_{e_1} = T_{e_2}$, but instead we got:

$$T_{e_1} = 0.0003550 \qquad T_{e_2} = 0.00001596496$$

As we can see, T_{e_1} is an order of magnitude greater than T_{e_2}. Even with all the approximations made, this is quite an odd result, which made us think that the time model we have defined does not properly fit method 1.
We must recall that to apply method 1 we need to run *two* simulations: *Simulation* ***1*** where $host_B$ is running an instance of *ipt_geofence* and *Simulation* ***2*** in which $host_B$ doesn't. From these two we obtain $pcap_1$ and $pcap_2$ and for each packet i in each `.pcap` file we calculate $T_e(i) = RTT1_i - RTT2_i$ and then we calculate the average of all $T_e(i)$, finally finding T_e.
Let's analyze more carefully this process. In principle, it should work because we have implicitly assumed that:

- $RTT1_i = T_{A2Q} + T_e(i) + T_{G2A}$
- $RTT2_i = T_{A2Q} + T_{G2A}$

and banally $RTT1_i - RTT2_i = T_e(i)$. While this time model works fine for $RTT1_i$, it does not for $RTT2_i$: there is **neither** *ipt_geofence* **nor** NFQUEUE in *Simulation* ***2***. T_{A2G} and T_{G2A} don't make any sense there.
We need to define the time intervals that make up RTTs more granularly to try to understand what actually happens when we apply method 1.

[3] maximum number of connections per second that *ipt_geofence* can handle without losing packets

Let's define:

- T_{A2B} as the average time a packet takes to go from NIC_A to NIC_B
- T_{B2PR} as the average time a packet takes to go from NIC_B to PREROUTING
- T_{PR2Q} as the average time a packet takes to go from PREROUTING to NFQUEUE
- T_{G2PS} as the average time a packet takes to go from *ipt_geofence* to POSTROUTING
- T_{PS2A} as the average time a packet takes to go from POSTROUTING to NIC_A

With these new definitions we can say that:

- $RTT1_i = T_{A2B} + T_{B2PR} + T_{PR2Q} + T_e(i) + T_{G2PS} + T_{PS2A}$
- $RTT2_i = T_{A2B} + T_{B2PR} + T_{PS2A}$[4]

and we have $RTT1_i - RTT2_i = T_{PR2Q} + T_e(i) + T_{G2PS}$.
Maybe this is why $T_{e_1} > T_{e_2}$, since method 2 calculates exactly T_e and not $T_{PR2Q} + T_e + T_{G2PS}$.
Anyway, to formulate more consistent considerations about why these two values differ so much, further analysis and investigation are needed.

6 Conclusions

We have successfully found a way, even though not so straightforward, to calculate the maximum number of connections *ipt_geofence* is able to handle without (eventually) dropping new packets on a commodity hardware.
While this is probably of no interest for standard *"home"* networks, it might be a crucial aspect if *ipt_geofence* runs on an *edge* node in a more crowded network; moreover, some users might find relevant to accept new connections when the queue becomes full, instead of automatically dropping them. This can be a cue for a new feature that allows the user to decide how to manage connections when NFQUEUE is full.
We have noticed also that the hardware on which *ipt_geofence* is running might improve its performance by non-negligible factor. It would be interesting to carry out further analysis on top tier hardware and possibly discover an upper bound to the maximum number of connections per second that *ipt_geofence* can handle without losing any new one.

[4] note that we should also add the time to go from PREROUTING to POSTROUTING but it should be negligible

Appendix

- *ipt_geofence* official repo: https://github.com/ntop/ipt_geofence
- python script to analyze `.pcap` files: https://github.com/frenzis01/pcap_stats_module
- python script to forge packets: https://github.com/yuricaprini/ntgenpy

Authors

Yuri Caprini

- y.caprini@studenti.unipi.it
- https://github.com/yuricaprini

Salvatore Guastella

- s.guastella2@studenti.unipi.it
- https://github.com/salvogs

Francesco Lorenzoni

- f.lorenzoni4@studenti.unipi.it
- https://github.com/frenzis01