IP Agnostic Real-Time Traffic Filtering and Host Identification Using TCP Timestamps

Georg Wicherski, Florian Weingarten, and Ulrike Meyer
Department of Computer Science
RWTH Aachen University
Aachen, Germany
gw@mwcollect.org, {weingarten, meyer}@itsec.rwth-aachen.de

Abstract—**In this work, we describe and evaluate the design and implementation of natfilterd, a flexible and lightweight extension of the Linux netfilter packet filter framework, which enables us to identify hosts completely independent of IP addresses by taking advantage of certain characteristics of TCP timestamps. As an immediate consequence, not only can we count hosts behind a NAT gateway but block TCP traffic from single hosts without blocking the gateway itself. Our work extends ideas from Bursztein, which we improve in terms of performance as well as matching quality and usability in practice. A theoretical runtime of O(log(n)) for matching packets against a database of n hosts is achieved. We empirically verify this result and conclude that our approach scales extremely well and is therefore suitable for at least medium-scale networks of a few thousand hosts.**

I. Introduction

Network address translation (NAT) allows multiple hosts of a private network to share a single public IP address. Originally, NAT was designed to be a short-term solution to the problem of IPv4 address depletion [1]. Today, NAT has become an essential requirement for the operation of the Internet. NAT deployment ranges from private home users with a few machines behind to large intranets connected to the Internet via NAT gateways.

Apart from addressing the address depletion problem, NAT is often advertised to be a privacy measure as it seems to hide the size and topology of the internal network. This, however, has been shown to be ineffective. There are various simple mechanisms that allow for the detection of the presence of a NAT, e.g. based on observing the TTL field in the IPv4 header [2]–[5] or on observing protocols that use fixed source ports [6], [7]. Moreover, various approaches to counting machines behind a NAT have been proposed in the past. Most of these proposals (e.g. [8], [9]) are based on observing header fields in the IPv4 or TCP headers that stay untouched when the packet passes a NAT. Examples for these fields are the ID field of the IPv4 header or the timestamp value of the TCP header. One of the most evolved approaches to counting machines behind NAT is due to Bursztein [10], [11]. It is based on the observation that the boot time of a machine together with the frequency of its TCP timestamp clock uniquely identifies it.

The wide deployment of NAT has also introduced problems for intrusion detection systems (IDS). In particular, IDS struggle with the fact that with NAT, a single IP address no longer uniquely identifies a single machine. Assume a machine behind a NAT is the source of malicious traffic. If the IDS simply blocks all traffic originating from this machine's IP address, it also blocks all non-malicious machines behind the same NAT. In order to be able to block only traffic originating from the malicious machine, the IDS needs to be able to identify the different machines behind the NAT in real time. This is the problem we address here.

In this paper, we propose natfilterd, a novel algorithm for identifying machines behind a NAT in real-time. natfilterd allows incoming IP packets to be matched to a host in real-time and thus allows an IDS to selectively block TCP traffic originating from a particular machine behind a NAT. natfilterd builds on Bursztein's idea but replaces his simple but error prone matching algorithm with an algorithm based on linear regression. This results in an algorithm with a very high matching quality. In addition, we introduce the concept of rate classes to improve the runtime of the matching algorithm by reducing the number of known hosts against which a newly arriving packet needs to be matched. We implemented natfilterd as a flexible and lightweight extension to the Linux netfilter packet filter framework. Our evaluation of the accuracy and performance of natfilterd confirms its suitability for real-time traffic blocking. To the best of our knowledge, natfilterd is the first real-time capable traffic filtering tool based on TCP timestamps. Other potential applications of natfilterd include the detection of connection sharing by Internet Service Providers, estimating the number of distinct servers behind a load balancer, or tracking machines using multiple IP addresses.

The rest of this paper is structured as follows. In Section II we summarize related work on NAT detection and counting machines behind a NAT. In Section III, we provide a high-level overview on how natfilterd works. This is followed by a detailed description of the algorithms implemented in natfilterd including a brief analysis of their complexity in Section IV. Finally, in Section V we provide benchmarks for both the matching quality and performance.

II. Related work

In the past, various different approaches to detect the presence of a NAT gateway or to identify and count machines behind a NAT gateway have been proposed. In the following we briefly describe the main ideas of these approaches.

Approaches to NAT detection are mostly based on the *time to live* (TTL) field in the IPv4 header [2]–[5]. This field is initialized by the sender of an IP packet to some fixed value (typically a power of 2) and decremented by every forwarding node on the routing path. Thus, if traffic coming from a supposed end-host is observed and the TTL value is not a power of 2, the observer can conclude that there is a NAT gateway in between. Note that, most operating systems support changing the default TTL value such that detecting a NAT with this simple aproach can be prevented. In [3] Phaal describes using TTL values for NAT detection while performing offline analysis of flow data containing captured packet headers. A similar approach is taken by Krmicek et al. [4]. Maier et al. [5] use TTL values to estimate that more than 90% of the DSL connections of a major European Internet service provider use NAT gateways.

Another approach to detecting the presence of a NAT is observing protocols whose client implementations typically use fixed source ports. NAT gateways often have to rewrite source ports and can therefore be detected if the source ports deviate from the standard ones. For example, Armitage [6] used log files of different online gaming servers (e.g., Quake III Arena) to count the number of clients who connected from a non-default port, thus estimating the amount of Quake III Arena players who are connected to the Internet via a NAT gateway. The same idea has been used to estimate the number of worm infections and other malware [7], for example for the W32.Witty.Worm [12], whose outbound connections use a fixed UDP port of 4000.

Various approaches for identifying or counting hosts behind a NAT have been proposed. Most of these approaches are based on the observation that parts of an IP packet stay untouched by a NAT. E.g. Bellovin [8] observed that many operating systems implement the ID field of the IPv4 header used to reassemble fragmented IP packets as a simple incremental counter. Consequently, consecutive IPv4 packets transmitted by a single host have sequential IDs. Assuming that hosts do not generate too much internal traffic (which a passive outside attacker could not observe), this idea can successfully be used to distinguish traffic from different hosts, even if the observed source IP address is the same. Note that, some operating systems (for example OpenBSD [13]) can use random values in the *identification* (ID) field and thus render Bellovin's approach useless. Bellovin suggests that NAT gateways could rewrite the field by themselves without breaking any functionality. Also note that IPv6 headers do not contain such a field, so this approach is restricted to IPv4.

Beverly [14] implemented a Naive Bayes classifier, a simple statistical learning method based on Bayes' theorem, to perform maximum-likelihood estimations on host operating system classification. He observes IPv4 header information, namely the TTL value, the *Do not Fragment* (DF) flag, and different window sizes from the TCP header. Beverly mentions NAT host counting as one possible application of host identification. However, his approach is performed offline on captured packets.

Besides the above "low level" approaches, which only look at IP headers and TCP headers, there are also a few ideas which utilize deep packet inspection and thus operate on the application layer. For example, Zhao et al. [15] and Bi et al. [16] inspect the payload of packets which belong to connections of instant messaging clients. They argue that this is a suitable approach to host counting since most users run only one instance of an instant messaging client on each machine. This approach can (in contrast to the "low level" approaches) also work if clients use application layer proxies such as HTTP or SOCKS proxies instead of NAT. Cohen [17] also suggests to use data from application layer payload such as HTTP user agent strings, HTTP referrers, HTTP session IDs and cookies, user names, and chat pseudonyms. Obviously, those approaches are not suitable for generic traffic filtering.

Using the timestamp value of the TCP header is widely used for passive OS fingerprinting and has been used in tools like `p0f` [18] and `nmap` [19] for years. For example, McDanel [20] describes how to remotely determine the uptime of a system by observing only two TCP packets with timestamps. He also mentions the idea to use timestamps for detecting load-balanced environments. As far as we know, the first ones to suggest the use of TCP (or ICMP) timestamp values for NAT host counting were Kohno et al. [9]. They identify hosts by determining local clock skews. Besides host counting, they also suggest to use this idea to distinguish honeypots from real systems and for deanonymizing anonymized data sets. Furthermore, Zander and Murdoch [21] use similar clock skew based host identification methods to identify hidden services in anonymity networks like TOR.

Bursztein [10], [11] described a novel approach to identifying hosts based on a combination of the boot time of a machine and the frequency of its TCP timestamp clock. He wrote a proof of concept implementation in the C programming language, using the packet capture library `libpcap`. This proof of concept implementation tries to count (but not filter traffic of) hosts behind a NAT gateway.

Like the work of Bursztein, our work is based on identifying hosts based on their boot time and the frequency of its TCP timestamp clock. However, we replace Bursztein's very simple (but error-prone) matching algorithm with an algorithm based on linear regression. This improves the matching quality. Furthermore, we increase the performance of the matching algorithm in practice by introducing the concept of *rate classes*. The resulting performance gain allows us to aim for real-time traffic filtering while Bursztein's approach was suitable for host counting only. As far as we know, our work is the first publicly available benchmark of performance and reliability of a TCP timestamp based, IPv6 capable, real-time traffic filtering tool.

III. Methodology

TCP timestamps were introduced in RFC 1323 [22]. They are specified as an optional 10 byte block inside the TCP header's option field. Those 10 bytes contain a 32 bit subfield, titled `TSval`. According to the RFC, this value is defined as follows.

> The timestamp value to be sent in `TSval` is to be obtained from a (virtual) clock that we call the "timestamp clock". Its values must be at least approximately proportional to real time, in order to measure actual RTT.

Mathematically speaking, `TSval` is an affine-linear function $t(s) = a{\cdot}s{+}b$ in time (ms), where a is the slope, i.e., the tick scale of the timestamp clock and b is some initial value. In practice, s is usually represented as a UNIX timestamp (in milliseconds), i.e. the number of milliseconds elapsed since midnight of January 1, 1970 (UTC). The RFC suggests that the scale a should be chosen between 0.001 (one tick per second) and 1 (one tick per millisecond). However, the exact value of a is implementation specific. Furthermore, operating systems usually reset the value of `TSval` to zero at boot time, thus effectively setting b to $-a \cdot s_0$ where s_0 is the time of booting the system.

Our work is based on the assumption that the pair (a, b) uniquely identifies a machine (until rebooted), as already postulated by Bursztein [10]. With *machine*, we refer to a single running system (physical or virtual), possibly behind a NAT gateway. Of course, this assumption is only valid assuming that no two machines with the same operating system (same tick scale) are booted at the same time.

Even when using network or port address translation, a 4-tuple $(\text{src}_{\text{addr}}, \text{src}_{\text{port}}, \text{dest}_{\text{addr}}, \text{dest}_{\text{port}})$, containing source IP address, source port, destination IP address, and destination port uniquely identifies (one direction of) a TCP connection between two machines as long as no RST/FIN packet is received and no connection timeout occurred. We therefore simply refer to such tuples as *(active) connections*. If a connection becomes inactive (i.e., terminates or times out), the same tuple can identify a new active connection.

The basic idea of our solution is as follows. For every incoming TCP packet observed (regardless of whether it is an IPv4 or IPv6 packet), add the pair (s_i, t_i) of current system time s_i and TCP timestamp t_i to a list for that particular TCP connection. As soon as the connection terminates (i.e., a RST/FIN packet is received), a least-squares linear regression function is computed for the points $(s_i, t_i)_{1 \le i \le m}$ collected for this particular connection. This will yield parameters (a, b) such that the sum of squares $\sum_{i=1}^{m} \epsilon_i^2$ of the errors $\epsilon_i = |t_i - (as_i + b)|$ is minimized. We then create a new host entry in our database with a unique id and associate the tuple (a, b) to it. One could therefore think of hosts as "lines", defined by functions $t(s) = a \cdot s + b$. In fact, this is the only information stored for each host. In particular, we do not need any IP addresses or ports of the packets, thus machines which use multiple IP addresses can also be identified.

Whenever a new TCP packet is received, we first try to *match* it against existing host entries. This can be done by computing the (euclidean) distance of the point (s, t), defined by that packet, to the lines of all known host entries. If the distance to one of those lines is below a certain threshold δ_{boot} (see next Section), the packet is associated to that host. If we are unable to match the packet, it is added to the list of that connection as described in the previous paragraph.

An alternative approach would be to *match connections* instead of *matching single packets*. We could collect packets and wait until a final RST/FIN packet is received before performing the matching. Since in this case, more data would be available, the matching would probably be more reliable. However, we want to identify machines in real-time for traffic filtering and therefore need to match on a per-packet basis.

The above approach is not very efficient because every new packet would have to be tested against every known host. To improve efficiency, we map hosts into *classes*. The tick scales a, from now on also referred to as *(clock) rates*, are rounded to two digits after the decimal point (which will result in a granularity of $0.01ms$). We apply this rounding to the rates of all hosts and group hosts with the same rounded rate into classes, which we call *rate classes*. Observe that the corresponding lines of all hosts of the same rate class are parallel (since the rate class is the slope of those lines). We store the hosts of a particular rate class in an ordered way such that consecutive hosts correspond to consecutive lines[1]. If we want to match a new packet, we iterate over all known rate classes and in each iteration, create an "artificial line", defined by the current rate class and the point defined by that packet. The distance of this artificial line to its (at most two) neighboring lines in that rate class is then computed. If any of those artificial lines is "close enough" to the line of some known host, the packet is associated to that host[2].

This approach is far more efficient in practice, assuming that the number of hosts becomes large but the number of

[1]For two parallel lines defined by $t_{1,2}(s) = a \cdot s + b_{1,2}$ with positive slope $a > 0$, we say that t_1 *is smaller than* t_2 iff the s-axis intersection (i.e., the unique zero) of t_1 is smaller than that of t_2. Equivalently, $t_1(s) > t_2(s)$ for some (and thus all) values s. Basically, t_1 is "above" (or "to the left of") t_2.

[2]We explain later what we mean by "close enough".

different rate classes remains low (which seems likely, since there is only a small number of tick scales implemented in operating systems and we only respect measurement errors of at most $0.01ms$). The main reason why the rate class approach is faster is because we only compare with at most two lines (and retrieving those two lines from the database is efficient).

IV. Implementation details and analysis

To implement our idea, we used the C++ programming language to create a software package called natfilterd. We use libnetfilter-queue from the netfilter project (http://www.netfilter.org). This way, we can use the iptables user interface to add certain *queue* rules to an existing firewall setting which causes the kernel to hand over TCP packets to natfilterd for further processing. At the end of this processing, natfilterd decides whether a packet should be *dropped* or *accepted* (i.e., returned to the kernel). Additionally, packets can be *marked* with a certain label in order for later firewall rules to identify them and perform further processing. For the computation of the least-squares linear regression function, we use the GNU scientific library (gsl), especially gsl_fit_linear.

As already sketched in the previous section, the main part of our implementation consists of two algorithms: AddHost and MatchPacket. These algorithms utilize the following data structures.

- T is a hash table mapping active TCP connections (i.e., 4-tuples $(\text{src}_{\text{addr}}, \text{src}_{\text{port}}, \text{dest}_{\text{addr}}, \text{dest}_{\text{port}})$) to lists of timestamp tuples $(s_1, t_1), (s_2, t_2), \ldots$ of current system time s_i and TCP timestamp value t_i.
- In the database of known hosts, we store the lines $t(s) = a \cdot s + b$ corresponding to the known hosts in the form (a, s_0) where $a > 0$ and s_0 is the solution of the equation $t(s) = 0$, i.e., the unique s-axis intersection of the line. If we know a and some point (s', t') on the line, we can compute this value as $s_0 = -\frac{t'}{a} + s'$. As already mentioned in the previous section, the s-axis intersection s_0 of a line corresponds to the boot time of the machine.
- R is a list of (non-empty) rate classes. Each rate class consists of a (rounded) rate a and a self-balancing binary search tree, indexed by s-axis intersections s_0, containing host ids as nodes. To each host id, we associate a filter policy (e.g., accept, drop, mark, ...).
- δ_{boot} is the threshold mentioned in the previous section. We assume that machines are booted at least this many milliseconds apart from each other. Otherwise, they are identified as the same, given they are in the same rate class. We use a default value of $2ms$.

Every time a TCP packet with a valid TCP timestamp is received, we call MatchPacket(s_i, t_i). The timestamp tuple (s_i, t_i) of current system time s_i and timestamp value t_i is added to the list $T(c)$, if that list already exists. If the packet was a RST/FIN packet, we call AddHost$(T(c))$ and delete the list of timestamp tuples $T(c)$ that we collected for connection c. New lists of timestamp tuples are only created if MatchPacket fails to identify the host. This way, we ensure that new connections are only processed if they belong to a previously unknown host. Once a host is in the database, we never call AddHost for its connections again.

```
MatchPacket($s, t$) – Matching a packet to a known host
  min ← $\infty$
  for all non-empty rate classes in $R$ do
    Compute parameters $(a, s_0)$ for the line defined by
    point $(s, t)$ and (rounded) rate $a$.
    Get the neighbours of $(a, s_0)$, i.e., the lines directly
    to the left and directly to the right.
    for all neighbour ∈ neighbours do
      if dist($(a, s_0)$, neighbour) < min then
        min ← dist
        host ← neighbour
      end if
    end for
  end for
  if min < $\delta_{\text{boot}}$ then
    return host
  else
    return unknown
  end if
```

The outer loop has $|R|$ and the inner loop has at most two iterations. The only step which requires non-constant time is to get the (at most) two neighbours of the current line. Since we store hosts in a search tree (indexed by s-axis intersection of the corresponding lines), this can always be done in $O(\log(n_r))$ where $n_r \leq n$ is the number of hosts in the current rate class. Therefore, MatchPacket runs in time at most $O(|R| \log(n))$.

```
AddHost($T(c)$) – Adding a host to the database
  Compute a least-squares linear regression function
  $t(s) = a \cdot s + b$ for the set of points $T(c)$.
  $a$ ← round($a$)
  Compute the line $(a, s_0)$ defined by the (rounded)
  rate $a$ and the last point in $T(c)$ as reference.
  if rate class with rate $a$ already exists then
    Get the neighbours of $(a, s_0)$.
    if $(a, s_0)$ is closer than $\delta_{\text{boot}}$ to one of its neigh-
    bours then
      return
    else
      Add new host to the search tree of the rate class.
    end if
```

```
else
    Add rate class to $R$ and add a new host to the search tree of the new class.
end if
```

Note that we do not use the constant b from the linear regression at all but only the slope a. Since we lose precision by rounding a in the AddHost algorithm, this value b would probably introduce too many errors. Therefore, we pick a reference point and create a new line with slope a which this reference point lies on. Experiments showed that using the most recent point as reference seems to be a good choice. Another idea would be to use the center of gravity of the points in $T(c)$, i.e., $\frac{1}{|T(c)|}\sum_{i=1}^{|T(c)|}(t_i, s_i)$. Note that the center of gravity always lies on the original line (before rounding the slope a).

Assuming constant errors for the points in $T(c)$, like gsl_fit_linear does, the linear regression function can be computed in time linear in the number of points $|T(c)|$. If there already is a host in the same rate class, we perform a search tree lookup as before which again is in $O(\log(n_r)) \subseteq O(\log(n))$. If the new host is not identified as an already existing host, we add it to the search tree, which can also be done in $O(\log(n))$. Therefore, the runtime of AddHost is bound by $O(|T(c)|) + O(\log(n))$. In practice, $|T(c)|$ can be assumed constant by setting an upper limit for the number of points to store for each connection.

Since system load and network latency variations can introduce too many different (non-constant) errors, we compute the Pearson correlation coefficient of the dataset. If the value is below a certain threshold (which indicates that there is no good linear correlation), we do not add the host at all. We use the gsl_stats_correlation function from the GNU scientific library for this.

As mentioned before, the number of rate classes occurring in practice is very low because there are only a few different clock rates implemented. Summarizing, if we assume a constant maximal number of rate classes and a constant maximal number of packets to collect for each connection, we have a theoretical runtime of $O(\log(n))$ for each incoming TCP packet with valid timestamp and an additional $O(\log(n))$ for each RST/FIN packet.

Since AddHost is called exactly once for each host in the database, this results in an overall complexity of $\sum_{i=1}^{n} O(\log(i)) \subseteq O(n\log(n))$ for all calls to AddHost in total.

Considering a complexity of $O(|R|\log(n))$ for packet matching, the question comes to mind why not to choose very small values of $|R|$. Recall that the value of $|R|$ highly depends on the rounding of rate values in such a way that a less accurate rounding will result in a smaller value of $|R|$. This will in fact increase performance, but at the cost of decreasing the matching quality. Our experiments showed that rounding to two digits after the decimal point is a good trade-off which results in about 100 different rate classes (since the rounded rates almost always lie in the range from 0.01 to at most 1.00) while at the same time guaranteeing a satisfactory matching quality.

V. Benchmarks

For evaluating the matching quality and performance of our implementation, we run natfilterd on a dedicated machine which received all ethernet traffic, incoming and outgoing, of our research institute. This is realized by configuring a mirror port on our local network switch. All tests are performed on a Debian GNU/Linux based system, running a 64 bit version of kernel 2.6.38 on a Intel Core i7-2600 CPU at 3.40 GHz and 16 GiB of RAM. The network consists of 30 machines with static IP addressing. We did not control the boot time of the hosts.

Recall that by *host*, we refer to a host entry in our database. The running (physical or virtual) system which belongs to a host entry is called a *machine*.

A. Matching quality benchmarks

Our algorithms assume constant errors in the timestamp data and we therefore assume that a high jitter (packet delay variation) will drastically reduce the reliability of our matching. To test this assumption, we perform several tests with artificially introduced delay variation. To define a notion of "quality", we differenciate between three kinds of matching errors: *unknown matches*, *failed matches* and *outdated matches*. The matching algorithm returns an *unknown match* if it was unable to associate the packet to a host entry in the database at all. If a database entry for this machine exists, this is an error. Furthermore, packets could be matched to the wrong existing host entry. Since checking if a host is "wrong" requires a unique identifier, we ran our tests in a subnetwork with static IP addressing. If a packet from IP x is matched to a host which was added because of packets from IP y and $x \neq y$, we call this kind of error a *failed match*. In our tests, we observed that there are some IP addresses for which there exist several host entries. This could be due to rebooting, clock skews or errors in one of our algorithms. If a packet from IP x is matched against a host which was also added because of IP x but was not the most recent host for this IP, we call this an *outdated match*. The motivation is as follows: If there are several host entries which were added because of the same IP address, this means that the machine was rebooted. If the matching results in an "old" host entry belonging to this machine, this is consider a matching error.

For each IP address in our network, we count the number of observed packets and the number of packets which lead to matching errors. The fraction of packets without errors is called the matching *quality* of an IP address. For the

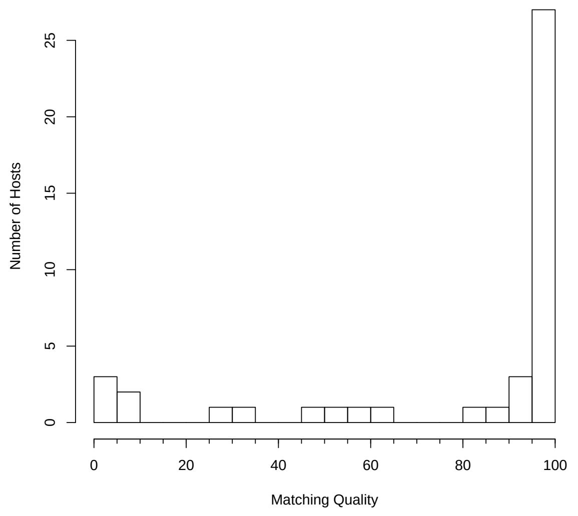


Figure 1: Quality distribution in the jitter 500 benchmark.

overall quality of one measurement, we compute an *average quality* (average of the qualities of all IP addresses) and a *total quality* (ratio of total number of packets without errors to total number of packets). The *total quality* is therefore weighted by the number of packets, i.e., the quality of machines with many packets will be more influential than those with less packets.

Table I shows the results of our benchmark. The quality columns show the overall qualities of the measurements (in percent, truncated to two decimal digits, no rounding), calculated based only on the respective criteria. The last column shows the quality with respect to all three error types combined. The first value is always an average value and the second (in parantheses) is a total value as described in the previous paragraph.

As one can see in Column 2 of Table I, the problem with multiple host entries per IP gets worse with increasing jitter. Interestingly, the number of outdated matches does not seem to increase significantly. Failed matches occur very rarely, even with high jitter. Our tests showed that the ratio of failed matches to the total number of errors is always less than 1%, which we consider very good. The overall quality is always at least 79.65%.

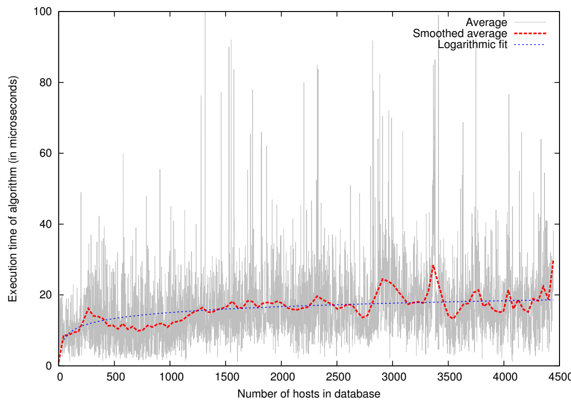
Figure 1 shows the distribution of qualities in the jitter 500 benchmark, which was the one with the worst overall quality. One can see that the matching quality of the majority of machines is within the range of 95% to 100% and a few outliers are responsible for the bad overall quality.

B. Performance benchmarks

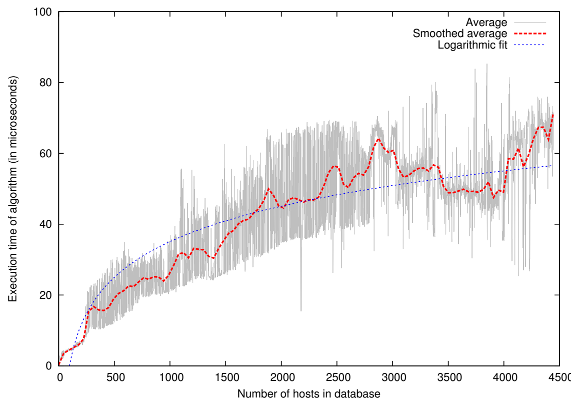
For performance evaluation, we implemented time measurements for the two algorithms AddHost and MatchPacket. We ran five tests, each lasting one day. The results presented in this section are averaged over all of those tests. We configured the kernel of our test machine to pass all traffic to natfilterd, so the number of host entries consists of our internal hosts as well as hosts on the Internet which answered our outgoing requests. In our tests, we added more than 4400 hosts in total (recall that with "host" we mean a host entry in our database, not necessarily a distinct machine). Figure 2 shows the results. Since the averaged values still fluctuate heavily, we decided to present them in a smoothed way. The blue dashed line shows a logarithmic fit to the raw values, corresponding to the theoretical complexity of $O(\log(n))$. Surprisingly, the logarithmic growth for the AddHost runtime is quite small. Furthermore, the runtime of MatchPacket in our evaluation is almost always below $80\mu s$, which is much better than we initially expected.

C. Discussion

Our practical performance analysis of natfilterd confirmed the theoretical complexity of $O(\log(n))$. We have shown that even with up to 4500 host entries, natfilterd requires less than $80\mu s$ to match an incoming packet to a host in the host database. These performance results were produced on reasonably up-to-date desktop hardware. We therefore



(a) AddHost



(b) MatchPacket

Figure 2: Performance of AddHost and MatchPacket.

Jitter	Host entries per IP	Quality (unknown)	Quality (fail)	Quality (outdated)	Quality (all)
$0ms$	1.61	92.98 (99.93)	99.99 (99.99)	99.06 (99.85)	**92.04** (99.78)
$10ms$	2.36	88.69 (99.43)	99.99 (99.99)	99.73 (99.58)	**88.43** (99.01)
$100ms$	2.55	91.37 (98.69)	99.99 (99.99)	95.40 (85.52)	**86.77** (84.21)
$250ms$	4.10	90.28 (98.39)	99.99 (99.99)	90.73 (98.18)	**81.01** (96.57)
$500ms$	5.60	83.41 (98.12)	99.94 (99.99)	96.29 (98.31)	**79.65** (96.42)
$1000ms$	4.96	84.54 (98.29)	99.99 (99.99)	98.13 (99.90)	**82.66** (98.18)

Table I: Matching quality benchmark results

conclude that our solution scales extremely well and can at least be deployed in small to medium-sized networks of a few thousand machines.

Our main application in mind is real-time TCP traffic filtering. Services and applications running on top of TCP will be severely corrupted as soon as a high percentage of related packets are dropped. As a consequence, traffic filtering can already be done successfully even if some percentages of the packets that should be dropped are not dropped. Our practical analysis has shown that even with unrealistically high jitter of $1000ms$, unknown matches (which can lead to packets not being blocked although they should be) occur in natfilterd with a probability of 16%. On the other hand, a few corrupted packets will typically not have any visible influence on a service or application. Failed matches (i.e., matching an incoming packet to a host that was added for a different machine) lead to packets being dropped for the wrong machines. Luckily in natfilterd, failed matches occur extremely rarely. We therefore conclude that natfilterd is a very suitable tool for TCP traffic filtering.

Another potential application of natfilterd is identifying or counting machines behind a NAT. For such an application to work reliably, the number of host entries per machine should be quite low. With (unrealistically) high jitter values, we had up to 5 entries per machine, while with lower jitter values the number of host entries per machine ranged around 2.5 entries on average. Regardless of the jitter values, however, we observed that the number of host entries per machine only increased after a few hours. As counting machines behind a NAT, NAT detection, or fingerprinting load-balanced environments does not require much time, we expect natfilterd to work reliably in these situations. As opposed to those applications, natfilterd does not seem to be the method of choice for tracking machines over longer time periods.

Recall that natfilterd identifies machines based on their timestamp clock rate and boot time. The larger the network, the higher the probability that two machines with the same clock rate are booted at approximately the same time. Therefore, we do not expect equally good matching results for very large networks as in our experimental setup.

There are several ways to potentially evade natfilterd. The most obvious one is disabling timestamps. However, this can greatly degrade TCP performance and is therefore unlikely to be used by any attacker. Another effective way to evade natfilterd is to use a separate TCP timestamp clock for each active TCP connection (rather than one for each host). To the best of our knowledge, OpenBSD is the only operating system that supports this feature out-of-the-box. It might also be possible to have the NAT gateway rewrite TCP timestamps of all routed packets transparently. However, this introduces additional load and memory requirements on routers, which are often constrained embedded devices. We are not aware of any manufacturers which actually implement this feature. As a consequence, in todays default setups, we will be able to successfully filter TCP traffic of single hosts behind a NAT gateway in real-time.

D. Conclusion and Future work

In this paper we introduced natfilterd, which allows for highly efficient TCP traffic filtering of single machines behind a NAT gateway. Our practical evaluation shows that natfilterd is reliable and real-time capable. In addition, our evaluation allows us to expect that natfilterd can be used for counting machines behind a NAT and fingerprinting load-balanced environments. For performance reasons, we chose to use the gsl_fit_linear algorithm from the GNU scientific library which is very fast, but assumes constant errors in the data. Our tests showed that non-constant errors (jitter) will decrease the matching quality. Therefore, it might be useful to look into other linear regression algorithms which are less prone to outliers in data. One possibility would be to use the RANSAC algorithm, which tries to decrease the impact of outliers when using least-squares fitting methods.

Acknowledgements

We would like to thank Bernd Kohler and Jens Hektor from the network operations center of RWTH Aachen University for their kind support in setting up our test environment. This work has been supported by the UMIC Research Centre, RWTH Aachen University.

References

[1] K. Egevang and P. Francis, “The IP Network Address Translator (NAT),” RFC 1631 (Standard), Internet Engineering Task Force, 1994.

[2] J. Postel, “Internet Protocol,” RFC 791 (Standard), Internet Engineering Task Force, 1981.

[3] P. Phaal, "Detecting NAT Devices using sFlow," http://www.sflow.org/detectNAT/, accessed 17 August 2011. (Archived at http://www.webcitation.org/610DAbCSQ).

[4] V. Krmicek, J. Vykopal, and R. Krejci, "NetFlow Based System for NAT Detection," in *Proceedings of the ACM Co-Next Student Workshop '09*, 2009.

[5] G. Maier, F. Schneider, and A. Feldmann, "NAT usage in Residential Broadband Networks," in *Proceedings of PAM'11*. Springer-Verlag, 2011.

[6] G. Armitage, "Inferring the Extent of Network Address Port Translation at Public/Private Internet Boundaries," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 020712A, July 2002.

[7] M. Abu Rajab, F. Monrose, and A. Terzis, "On the Impact of Dynamic Addressing on Malware Propagation," in *Proceedings of the ACM WORM'06*, 2006.

[8] S. M. Bellovin, "A Technique for Counting NATted Hosts," in *Proceedings of ACM IMW'02*. ACM, 2002.

[9] T. Kohno, A. Broido, and K. C. Claffy, "Remote physical device fingerprinting," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2005.

[10] E. Bursztein, "TCP Timestamp To count Hosts behind NAT," Phrack Magazine, issue #63, article 0x03-2, http://www.phrack.org/archives/63/p63_0x03_Linenoise_by_Phrack%20Staff.txt, Aug. 2005.

[11] ——, "Time has something to tell us about Network Address Translation," ENS Cachan, France, http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/Bur-nordsec07.pdf, Jul. 2007, accessed 17 August 2011. (Archived at http://www.webcitation.org/610Df989u).

[12] E. Chien, "W32.Witty.Worm," Symantec security advisory, http://www.symantec.com/security_response/writeup.jsp?docid=2004-032009-1441-99, Mar. 2004, accessed 17 August 2011. (Archived at http://www.webcitation.org/610NryOez).

[13] "OpenBSD Programmer's Manual – `pf.conf` manpage," http://www.openbsd.org/cgi-bin/man.cgi?query=pf.conf, accessed 14 September 2011. (Archived at http://www.webcitation.org/61gfvm21A).

[14] R. Beverly, "A Robust Classifier for Passive TCP/IP Fingerprinting," in *Proceedings of the PAM Workshop*, 2004.

[15] L. Zhao, M. Zhang, J. Bi, and J. Wu, "Detecting Private Address Space based on Application Layer Information," in *First IEEE Workshop on Adaptive Policy-based Management in Network Management and Control*, 2006.

[16] J. Bi, M. Zhang, and L. Zhao, "Security Enhancement by Detecting Network Address Translation Based on Instant Messaging," in *EUC Workshops*, 2006.

[17] M. I. Cohen, "Source Attribution for Network Address Translated forensic captures," *Digital Investigation*, vol. 5, no. 3-4, 2009.

[18] M. Zalewski, "p0f - Dr. Jekyll had something to Hyde - passive OS fingerprinting tool," http://lcamtuf.coredump.cx/p0f/README, accessed 17 August 2011. (Archived at http://www.webcitation.org/610DSppNO).

[19] G. F. Lyon, *nmap Network Scanning: The Official nmap Project Guide to Network Discovery and Security Scanning*. USA: Insecure, 2009.

[20] B. McDanel, "TCP Timestamping and Remotely gathering uptime information," Bugtraq mailing list, http://seclists.org/bugtraq/2001/Mar/182, Mar. 2001, accessed 17 August 2011. (Archived at http://www.webcitation.org/610DI2qlN).

[21] S. Zander and S. J. Murdoch, "An Improved Clock-skew Measurement Technique for Revealing Hidden Services," in *Proceedings of the USENIX Security Symposium*, 2008.

[22] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," RFC 1323 (Proposed Standard), Internet Engineering Task Force, 1992.

[23] M. Casado and M. J. Freedman, "Peering Through the Shroud: The Effect of Edge Opacity on IP-Based Client Identification," in *Proceedings of USENIX NSDI'07*, 2007.