POLITECNICO DI TORINO
Repository ISTITUZIONALE

GT: Picking up the Truth from the Ground for Internet Traffic

*Original*
GT: Picking up the Truth from the Ground for Internet Traffic / Gringoli, F., Salgarelli, L., Dusi, M., Cascarano, N., Risso, F.G.O., Claffy, K.C.. - In: COMPUTER COMMUNICATION REVIEW. - ISSN 0146-4833. - 39:5(2009), pp. 13-18. [10.1145/1629607.1629610]

*Availability:*
This version is available at: 11583/2281783 since: 2016-05-23T18:19:56Z

*Publisher:*
ASSOC COMPUTING MACHINERY

*Published*
DOI:10.1145/1629607.1629610

*Terms of use:*

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

*Publisher copyright*

(Article begins on next page)

14 October 2026

# GT: picking up the truth from the ground for Internet traffic*

F. Gringoli, L. Salgarelli, M. Dusi
Università di Brescia

N. Cascarano, F. Risso
Politecnico di Torino

K. C. Claffy
CAIDA

## ABSTRACT

Much of Internet traffic modeling, firewall, and intrusion detection research requires traces where some ground truth regarding application and protocol is associated with each packet or flow. This paper presents the design, development and experimental evaluation of `gt`, an open source software toolset for associating ground truth information with Internet traffic traces. By probing the monitored host's kernel to obtain information on active Internet sessions, `gt` gathers ground truth at the application level. Preliminary experimental results show that `gt`'s effectiveness comes at little cost in terms of overhead on the hosting machines. Furthermore, when coupled with other packet inspection mechanisms, `gt` can derive ground truth not only in terms of applications (e.g., e-mail), but also in terms of protocols (e.g., SMTP vs. POP3).

## Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

## General Terms

Experimentation, Measurement

## Keywords

Ground truth, application layer, transport layer

## 1. INTRODUCTION

The majority of research activities carried on under the umbrella of Internet traffic analysis requires the association of application and protocol ground truth information with traffic traces. Most mechanisms used today to link ground truth meta-data to Internet traffic traces roughly conform to one of the two following procedures. One approach is to create a trace manually by instantiating a realistic pool of applications on many machines. However, such captured traffic typically lacks characteristics that human behavior can induce. The second approach is to record traffic on a live network, and apply deep packet inspection (DPI – pattern-matching filters) to each packet's payload, usually complemented by port analysis. But DPI is ineffective when traffic is encrypted and ambiguous when different protocols exhibit similar signatures, and port-based analysis is rapidly becoming useless.

*This work was supported in part by a grant from Cisco Systems, Inc.

This paper introduces "`gt`", a new mechanism to provide ground truth at the application level. The `gt` architecture is based on a client tool that, by monitoring a host's kernel, associates each packet flow with the name of its controlling application, and transmits the collected information to a back-end. The post–processing toolset "`ipclass`" analyzes the traffic captured at the network border by an independent probe and associates each flow with its application label, laying a reputable foundation for the establishment of ground truth for that flow. The tool works on many widespread operating systems, and it is freely available under an Open Source (BSD) license [1].

We evaluate the effectiveness of the toolset in two network environments, with the help of colleagues who consented to be monitored with `gt`. Our experiments show that the `gt` architecture can tag up to 99% of the bytes and 95% of the flows on all platforms, while consuming about 5% of the resources on 2GHz CPUs.

In order to derive ground truth both at the application level (e.g., attaching the "Firefox" or "Thunderbird" label to a given flow), and at the protocol level (e.g., attaching the "SMTP" or "HTTP" label), we include in `gt` a DPI-based mechanism. We show that the combination of the application label with payload inspection can significantly improve the accuracy of ground truth meta-data[1] compared to current approaches that rely solely on DPI.

The rest of the paper is organized as follows. Section 2 covers the related work. Section 3 presents the `gt` architecture, discussing the main technical aspects of its implementation. Section 4 describes our experimental testbed and Section 5 the results of our tests on `gt`. In light of such results, we further discuss two of the main design choices in Section 6, while Section 7 concludes the paper.

## 2. RELATED WORK

Payload inspection, if traces contain at least a portion of the payload, is one of the most popular techniques to establish a form of *protocol* ground truth [2, 3]. Port-based mechanisms are also used, especially for those working with publicly available traces whose payload has been entirely stripped [4, 5]. However, both port and payload-based techniques can only provide an estimate of the protocol being carried, in contrast with `gt` which unequivocally tags the flow with the application that generated it.

[1]Although ground truth should be, by definition, accurate, not all meta-data that specifies ground truth for Internet traffic traces is correct, since it is often derived with inaccurate means, such as port analysis or DPI.

Manual generation of traffic can provide ground truth at the *application* level. Recently Dusi et al. [6] used this technique to collect SSH-encrypted data, by capturing both clear-text and the corresponding SSH-tunneled sessions. Although this approach can deal with encrypted traffic, it cannot prevent system daemons or other background applications from generating their own traffic, which may not be possible to accurately tag.

Trestian, *et al.* [7] describe a heuristic that combines information freely available on the web (through Google) to retrieve the class of applications that generated a flow. The authors evaluate this heuristic on traffic flows collected on several Tier-1 networks, and correctly classify 60% of traffic whereas BLINC [8] classifies only 30%. Although the approach does not require running any daemon on the monitored host, it relies on external resources, namely Google, to keep current information about the web, as well as track DNS dynamics for services on dynamic IP address. Instead, our approach queries the name of the application directly on the machine that generated the flow.

Szabo et al. [9] offers a more advanced approach in application detection and tagging. The mechanism records part of the name of the application that generated each IP packet and embeds this information into the packet itself by means of a Router Alert IP option. Embedding ground truth information directly in IP packets poses some methodological problems. The "stamping" of each packet happens in real time, which might affect the packet capture process (e.g., packet inter-arrival times), lowering the precision of the recorded traces. Also, the mechanism cannot mark packets whose size is closer to the MTU (unless IP fragmentation is used), since it requires adding an option to the IP header. Furthermore their implementation is Windows XP-specific: porting to other OSes, even other Windows flavors, requires a considerable amount of work at kernel-level. The software also lacks a set of tools that allow capturing traffic and post-processing data in order to assign the ground truth to each connection; the tool provides *application* but no *protocol* information for a given flow. Finally, the tool has not been released to the public.

Canini et al. [10] recently presented a new platform, the Ground Truth Verification System (GTVS), that uses a combination of heuristics at different levels (host, flow, packet) to improve the quality of ground truth associated with packet traces. The GTVS platform does not (yet) include application labels guaranteed to be accurate, such as those provided by `gt`. Conversely, `gt` currently uses only a DPI–based mechanism to correlate application labels to protocol information, so it could benefit from the additional heuristics described in [10]. These two approaches are complementary; in future work we hope to evaluate whether in combination both tools perform better than either one alone.

# 3. THE `gt` ARCHITECTURE

The `gt` architecture has four elements: a *client daemon* that runs on all monitored hosts, retrieves from the kernel the name of the application that generated each flow, and sends this information to a remote back-end; a *packet capture engine*, which in our scenario is co-located with the network's border gateway to the Internet; a *database server* that collects the labels assigned by client daemons; and a set of *post-processing tools* called `ipclass` to label each flow with an application and optionally run other heuristic algorithms

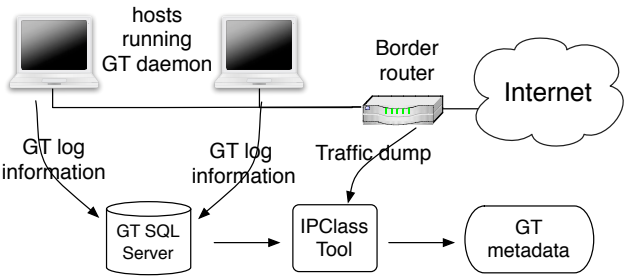


**Figure 1: A typical application scenario of `gt`.**

to pinpoint the flow's characteristics. Figure 1 depicts the scenario we used to test `gt`: a LAN with several hosts connected to the Internet via a border router, which records traffic between the monitored hosts and the Internet.

## 3.1 The `gt` client daemon

The main functionality of the client daemon is to track changes in active network sockets, and collect and transmit to the database server relevant information about the applications that own the sockets. We designed the daemon around a user-space table that mirrors the active socket list handled by the kernel. A single-thread loop periodically synchronizes the user-space and kernel tables, logging changes to the database. The loop frequency is configurable, and its value affects both `gt`'s accuracy and the performance impact on the hosting machine: Section 5.1 explores this tradeoff.

The client currently compiles and runs on many different platforms. It runs as a service on Windows Vista, XP and 2003. It works in daemon mode on many Unix-like systems, including Linux distributions with kernels 2.4 and 2.6, Mac OS X 10.4 and 10.5, and FreeBSD 5 and 6.

### 3.1.1 Kernel polling engine

The kernel polling engine is the core of the client daemon. It samples at fixed intervals the state of active sockets as recorded by the kernel, reconciling such information with `gt`'s table in user-space, and assigning an application name to each socket. The output is a list of active sockets, identified by the classical 5-tuple (source and destination IP, port numbers and transport protocol), each labeled with the name of the application that owns it.

The logic that associates each socket with the process that owns it, and its corresponding "application label", is different depending on the operating system. Table 1 reports the techniques the client daemon uses to retrieve socket information from the kernel and track changes in user space, along with the privileges required to install and run the client. Only Mac OS X 10.4 requires administrator privileges, since it does not provide *sysctl* calls to access the lists of sockets owned by each process: one must directly parse kernel memory or copy it to user space before parsing. Note that Apple documentation discourages this approach, since the data structures may vary by kernel release, and Apple's development forum suggests to "popen" an official `lsof` process and parse its output, the execution of which proved too slow for our client daemon needs.

### 3.1.2 Communication module

To synchronize data transmission (to the database) with polling without taxing system resources, at the end of each

| OS | Socket info | Priv. |
|---|---|---|
| Linux 2.4/2.6 | proc filesystem | user |
| Mac OS X 10.4 | /dev/kmem (PPC)<br>virtual mem fcnt+sysctl (Intel) | root |
| Mac OS X 10.5 | libproc+sysctl | user |
| FreeBSD 5.x/6.x | sysctl | user |
| Windows XP/ Vista/2003 | IpHlpApi.dll library | user |

**Table 1: List of operating systems supported by the `gt` client daemon, along with the techniques used to retrieve kernel socket information, and the type of privileges required.**

loop the polling engine organizes the ground truth information in a temporary buffer. The engine transmits the temporary buffer to the database at configurable time intervals (five seconds in our tests).

The client daemon also implements a heartbeat protocol, informing the database server of its status. This allows `ipclass` to detect when the client daemon is active on each host, and correlate such information with the traffic traces. In case the client daemon might be suspended, moved, or otherwise made unable to reach the database, the communication module re-initializes the daemon, signaling to the database a clean re-start and transmitting all currently active sockets. Thus, `ipclass` will know that a recovery has occurred and behave accordingly.

## 3.2 Packet capture engine and database server

The `gt` architecture can be deployed with any packet capture engine. The only requirement is clock synchronization between the capture device and database server, which must be accurate enough to ensure correct correspondence between the first packet of a flow and its ground truth log within the database. Similar considerations hold for the database server: any relational database would do. We built the current version of the toolset around `tcpdump` and MySQL.

Ground truth information coming from all `gt` client daemons is written to the same database table. Each row contains the 5-tuple for each logged socket, plus log time, name of the application owning the socket and type of log event: *create*, *destroy*, etc., depending on what happened to the corresponding socket.

## 3.3 Post-processing: the `ipclass` toolset

The `gt` architecture includes a set of post-processing tools which we developed, collectively called `ipclass`, and wrote in Python. `ipclass` reconciles ground truth information contained in the database with the captured traffic traces. While in the TCP case we consider only those flows starting with a valid three-way-handshake, for UDP we reserve a flow in the hash table as soon as we observe its first packet.

When we observe the first packet of a flow, say with timestamp $t_0$, `ipclass` looks in the database for a flow with a log time $t_{log}$ close to $t_0$, in a time window that depends on the polling time and the reporting interval (i.e., the time between subsequent transactions between the `gt` client daemon and the database). If found, the entry will unequivocally identify the application that generated the flow, according to the `gt`-client recorded information.

### 3.3.1 Handling UDP traffic

UDP traffic requires special attention when reconciling ground truth information with traffic traces. Applications can use two kinds of UDP sockets: connected and bound. The former is managed by the kernel through the `connect` syscall and allows the application to send messages to a single destination IP address/port. The kernel maintains information about the 5-tuple for each connected socket. The latter is handled through the `bind` syscall and allows applications to specify the destination IP/port address each time they send messages through it. We call this type of socket "anonymous UDP", for which the kernel stores only the local IP address and port pair.

To deal with anonymous UDP sockets, `ipclass` initializes an internal database when it starts, with the information concerning all tagged anonymous UDP sockets. For each socket it extracts the IP address $H$ of the logging host, the log time $T_S = t_{log}$, the ephemeral port $P$, the application label $A$ and the log type event. For a "create" event, `ipclass` adds an entry for that host+port to the UDP internal database. A "destroy" event for the same host+port will add the ending time $T_E = t_{log}$ to its entry: this way `ipclass` will derive that during the time interval $[T_S, T_E]$ port $P$ on host $H$ was owned by application $A$. Therefore, all flows that share such host+port will be tagged as belonging to $A$.

### 3.3.2 Associating protocols to flows

Although `gt` by design associates a flow with an application, such information can also be used to derive accurate ground truth with respect to the application protocol. In `ipclass` we implemented a DPI module that applies a selected set of signatures, specific to an application, to each network flow. We start by inspecting each application recorded by `gt` (e.g. browsing source code, reading public documentation and observing its behavior) and compiling a list of protocols used by the application itself. We then use the public signatures from `l7filter` [11] to construct a signature list that matches our observations. `gt` returns the protocol associated with each flow by matching each flow with the subset of signatures that we associated with that application.

Our experiments show that this selective signature-based matching is accurate. For example, in our tests it allowed the accurate tagging of some of the Skype flows, for which DPI–alone matched the signature of the NTP protocol. However, while `gt`'s reported *application* label is 100% accurate by design, it may report an incorrect *protocol*, or not detect it at all if no signatures match. Section 5 expands on this point.

# 4. TESTBED SETUP

We evaluated `gt` on two campus network environments: at the University of Brescia (UNIBS) and the Politecnico di Torino (POLITO), whose abstract topologies are shown in Figure 1. We found a set of informed users consenting to participate in the experiment, and instrumented a set of hosts on the same subnet with the `gt` client daemon, asking participants to use the Internet as they usually do.

At UNIBS, we installed the `gt` client daemon on a dozen machines inside the campus, running a mix of Mac OS X, Linux and Windows operating systems, used by graduate students and faculty. We captured all the traffic generated by these hosts for six days, collecting about 18GB.

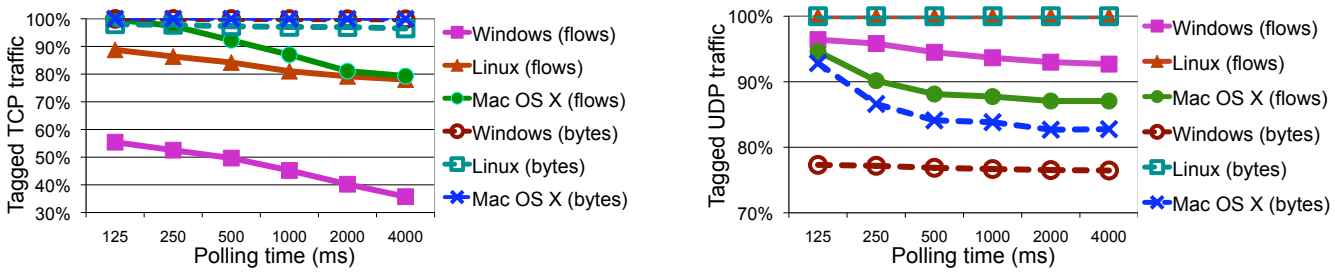


**Figure 2: Completeness for TCP (left) and for UDP traffic (right), standalone polling mechanism, with the service-based enhancement not enabled (see Section 5.1.1). Continuous line is for flows, dashed line for bytes.**

At POLITO, we installed the `gt` client on four real machines running Linux, Windows Vista and Mac OS X and on ten virtual machines running Windows XP, which executed two popular P2P WebTV applications (TvAnts, SopCast). We monitored for three days and collected 200GB of traffic.

In both environments we captured only traffic of the selected subnets when it traversed the university's upstream link to the Internet. Specifically we mirrored the upstream link to a machine running `tcpdump` [12] which captured full packets in pcap format. At UNIBS we ran vanilla `tcpdump` on a high-end, 2.4 GHz quad-core machine able to sustain a load that never exceeded a few tens of Mb/s. At POLITO we used an Endace capture card to sustain the high traffic load (more than 100Mbps) on the exit link. Internal counters on the operating system of the capturing machines indicated no packet loss during the capture. Socket information collected by `gt` on each machine was sent to a MySQL server running on a dedicated machine inside each campus' network. We used Network Time Protocol to maintain synchronization of the clocks of the capturing device and of the SQL server.

# 5. EXPERIMENTAL ANALYSIS

We evaluated the performance of `gt` with respect to two metrics. The first, which we call **completeness**, is the fraction of traffic produced by the instrumented end-hosts that `gt` is able to tag. A polling-based architecture is unlikely to catch every socket, so in our first experiments we try to maximize completeness given other performance constraints. To achieve a completeness approaching 100% while maintaining an acceptable CPU load on the monitored host, we included in `gt` a service-based mechanism (described by Baldi et al. [13] and also used by Karagiannis et al. [8]), that exploits the knowledge previously gathered by `gt` (while processing earlier flows) of which service is offered at given network coordinates (IP address and TCP/UDP port pair) to assign the same label to all sessions directed toward those coordinates.

Our second set of experiments examines how the combination of the application label and payload inspection can improve the **accuracy**[2] of ground truth meta-data for Internet traffic traces, as opposed to what obtainable using solely DPI-based mechanisms. Here "accuracy" is defined as the fraction of traffic (flows or bytes) that DPI correctly identifies as a particular application using a particular protocol. A DPI device can produce inaccurate output when no patterns match an observed flow, or multiple protocols match, or when a single match is found but it is wrong (false positive).

## 5.1 Completeness

The `gt` configuration parameter most relevant to completeness is the *polling time*, i.e., the time between two consecutive queries to the socket internal structures. Too short a polling interval will incur unnecessary overhead on the monitored host, but the longer the polling interval, the more sockets, and thus flows, we will miss. Our first experiments evaluated the effects of the polling time value on accuracy (completeness) and resource consumption (CPU load).

For these first experiments, we used three machines, a subset of those described in Section 4. One of them ran the 64-bit-version of Windows Vista, the other two ran Linux Gentoo 2.6 and Mac OS X Leopard 10.5.5 respectively. All machines were equipped with a 2GHz Intel Core 2 Duo processor and at least 3GB RAM. On each host we simultaneously launched several instances of `gt` with different polling time values and we let them run for about three hours. For each instance we calculated the percentage of CPU used (i.e., the ratio between the CPU time required by the given `gt` instance and the total CPU time) and the percentage of tagged TCP and UDP flows and bytes[3].

On all machines, the CPU load due to `gt` smoothly decreased as polling time increased. On all architectures, a 4-sec polling time caused a CPU load well below 5%. The load stabilized around 5% with a 1sec polling time for all architectures except for Linux, where the daemon induced a 12% CPU load. Shorter polling times led to increased load: with an interval of 125msec we registered a maximum CPU load of 50% on the Windows machine and around 20% on the other platforms. These results depend largely on the traffic generated by each host and how operating systems handle system calls.

Figure 2 (left) shows how the polling time affected completeness for TCP traffic, in terms of flows and bytes. Independent of the platform and polling times, `gt` tagged more than 99% of TCP traffic measured in bytes. In terms of flows, results thus far are less impressive: the fraction of tagged sessions was around 60%-80%, except for the Mac OS X case, which tagged over 90% of flows, at least with short polling intervals. It seems that the Mac OS X kernel maintains the link between the process name and its socket longer after the session has ended than other platforms, facilitating `gt`'s job.

[2]See footnote #1 of this paper.

[3]The total CPU load on each machine was always under 100% during the experiment.

| Application classes | Protocols (signatures) |
|---|---|
| Web-browsers (Safari, Firefox, etc.) | HTTP, SSL |
| Mail-clients (Evolution, Apple Mail, etc.) | HTTP, POP3, IMAP, SMTP, SSL |
| P2P-data ($\mu$Torrent, Transmission) | BITTORRENT, HTTP |
| Skype | SKYPE, HTTP |

**Table 2: Subset of protocols (and corresponding DPI signatures) used by a few sample applications from the UNIBS and POLITO datasets.**

Figure 2 (right) shows that for UDP traffic, `gt` performed better than for TCP: on Unix kernels, the completeness in terms of UDP flows always exceeded 87% (100% on the Linux machine), while it exceeded 78% in the TCP case. We believe this is due to anonymous UDP sockets that the kernel left indefinitely allocated. On the Windows platform, the percentage of tagged bytes stayed around 77% independent of polling time. We are trying to further investigate the source of this sub-standard result as follow-on work.

### 5.1.1 Approaching 100% completeness

In our experiments, a significant number of flows escaped `gt`'s observational power, especially on Windows, for which `gt`'s completeness for TCP *flows* was around 45% when using a polling time of one second. Further analysis showed that those flows were extremely short (on average, less than 200msec), which explains their small impact on `gt`'s completeness in terms of bytes.

To compensate for those flows without further decreasing the polling time, we paired `gt` with a service-based technique [8, 13]. Given an untagged flow, this technique looks for another flow that shares the network coordinates (destination IP address and TCP/UDP port pair) and was tagged by `gt`. If such a correspondence exists and is unique, we assume that both flows were generated by the same application.

We preliminary evaluated this method on the traffic generated by three machines at UNIBS (running Linux, Windows and Mac OS X, respectively) for which `gt` was configured with a one-second polling time, achieving over 95% of completeness across about 30,000 flows, i.e., `gt` successfully tagged 95% of flows produced by hosts, which corresponded to more than 99% in terms of bytes. A similar test led us to equivalent results for the POLITO trace, where completeness when using the service-based technique approached 100% in terms of both bytes and flows.

Although specific to the two traces we used, these results suggest that the service-based technique might effectively compensate for short flows generated by web browsers, mainly due to web-caching mechanisms, and for flows generated by P2P applications sending signaling traffic towards random ports.

## 5.2 Improving the accuracy of other ground truth mechanisms

Although `gt` labels refer to application names, and not application protocols, the knowledge of the `gt` label can be used to improve the accuracy of protocol ground truth obtained with other mechanisms, such as port analysis or DPI. Knowing the application that generated a given flow can, in fact, help narrow the *subset of possible protocols* that generated it. For example, the command-line `ftp` program

| | Flows | Bytes |
|---|---|---|
| UNIBS | 95.47% | 67.73% |
| POLITO | 79.31% | 58.79% |

**Table 3: Accuracy of ground truth meta-data derived with a DPI–alone mechanism.**

available on many BSD-like systems can use only the FTP-COMMAND and FTP-DATA protocols. A flow labeled by `gt` as `ftp` can then be tested with a DPI by considering only the patterns of these two protocols, thus reducing the risk of tagging it with an inaccurate protocol label.

Any technique that provides protocol-level ground truth estimates can take advantage of `gt` labels, and the protocol subsets they imply, to better ascertain the nature of a flow. In this section we evaluate the accuracy improvement `gt` potentially offers a DPI (payload inspection) technique, which is still the primary mechanism used to derive the ground truth about network traffic at the protocol level.

We started by processing the POLITO and UNIBS traces with a DPI mechanism that we implemented in `ipclass` based on all the signatures available in [11]: we call this method "DPI–alone". We then processed the same traces again with `ipclass`, but this time we used only the subset of signatures relevant to each flow, according to the `gt`-assigned application label as reported in Table 2. We call this mechanism "GT+DPI".

Given the absolute accuracy of the `gt` label at the application layer, the ground truth returned by the DPI–alone mechanism can be trusted only when its output matches that of GT+DPI. We therefore define "accuracy" of the DPI–alone mechanism as the fraction of traffic for which the protocol label assigned by GT+DPI uniquely corresponds to the one assigned by DPI–alone.

Table 3 presents the results of the experiment. On the UNIBS trace, DPI–alone correctly tagged 67.73% of the bytes: this means that `gt` had the potential to improve the accuracy of DPI by more than 32% (in bytes). `gt`'s ability to improve the flow-based accuracy of this trace was limited to only 4%, which means that the few flows where DPI failed carried substantial traffic volume in that trace. On the POLITO dataset, `gt` signaled inaccuracies of DPI–alone in more than 41% of the cases (in bytes) and in almost 21% in terms of flows, which were mostly connections generated by P2P applications.

We then extracted the fraction of flows and bytes for the applications where adding the `gt` label had the least and most significant impact for the two traces. Table 4 presents these results. As expected, for flows generated by applications that use mostly clear-text protocols (web browsers and mail clients), the output of DPI–alone is accurate (over

| | UNIBS | | POLITO | |
|---|---|---|---|---|
| | Flows | Bytes | Flows | Bytes |
| Web Browsers | 97.26% | 98.97% | 91.55% | 96.81% |
| Mail Clients | 92.37% | 90.46% | 90.00% | 35.82% |
| P2P (data) | 22.62% | 8.67% | 59.37% | 34.10% |
| Skype TCP | 4.64% | 16.34% | 4.99% | 9.20% |
| Skype UDP | 97.12% | 97.90% | 95.80% | 94.73% |
| P2P (WebTV) | N.A. | N.A. | 0.00% | 0.00% |

**Table 4: Accuracy of the DPI–alone approach: details.**

90%). The gt label is particularly useful for applications that use encryption or obfuscation mechanisms to hide their traffic, such as Skype and P2P file sharing applications. The gt label was also essential to ascertaining ground truth for the traffic generated by P2P video applications in the POLITO trace, such as TvAnts and SopCast, for which no public signatures were available. Finally, the results showed the accuracy of the L7filter signature for Skype UDP traffic, which the gt label confirmed was effective in more than 94% of the cases, in both bytes and flows. In contrast, the gt label also tells us that only a small fraction of Skype TCP traffic is accurately detected by L7filter's (DPI) signature, due to Skype's use of obfuscation and encryption [14].

# 6. DISCUSSION ON DESIGN CHOICES

We tested gt in this paper by decoupling the collection of ground truth knowledge, which is performed on end hosts by means of a client daemon, from the capture of traffic traces, which in our experiments was done on a border router (Figure 1). Although this scenario is quite typical, it is worth noting that the gt architecture fully supports other usage modes, such as a distributed deployment where each end host tags their flows and captures the traffic it produces without the help of a central node.

The second observation is related to the choice of implementing the tagging engine as a user-space polling daemon. During the development of gt, we also implemented kgt, a kernel version of the client tool for the Linux 2.6 architecture. Although this particular implementation allowed us to obtain 100% completeness even without the service-based enhancement, portability and extensibility issues led us to prefer the implementation of gt in user-space. Porting kgt to other UNIX-like kernel architectures such as Mac OS X is itself a daunting task, due to the radically different syscalls, and Windows would require a completely different approach, e.g., writing wrappers to intercept each socket operation, as anti-virus or anti-spyware tools do.

We opted to maintain only the user-space code, countering the resulting lower completeness with the service-based mechanism as explained in Section 5.1.1, while gaining an easily portable code base, and therefore greater applicability to diverse uses, environments, and research goals.

# 7. CONCLUSIONS

This paper presents the first implementation of gt, an open source [1] toolset that allows the capture of traffic traces with associated meta-data indicating what application actually sent each packet, a knowledge base essential to many Internet measurement research activities. By examining the TCP/IP stack of monitored client hosts, gt assigns application labels to traffic flows, allowing storage of ground truth with the trace without altering the statistical properties of the trace itself. In post-processing, such information helps in deriving not only meta-data on the generating application, but also the protocol used, by correlating the application label to the estimate of payload analyzers.

Experimental tests demonstrate the effectiveness of our approach, which can tag more than 99% of bytes and 95% of flows produced by monitored nodes, without significantly affecting CPU load, thanks to the combination of a polling mechanism with a simple service-based heuristic. Our experiments also demonstrate that gt can improve the accuracy of DPI–derived ground truth up to 85% (flows) for a critical application like Skype (on TCP), and up to 91% (bytes) for P2P applications. While these results are based on the applications present in our two sample datasets, we expect significant improvements for other applications that use encryption or obfuscation mechanisms.

gt can be useful to research activities that go beyond traffic classification. For example, it is being evaluated at the LIP6 laboratories in Paris as a tool to help the automated diagnosis of applications responsible for performance degradations in local networks.

We are currently working to further optimize gt's code in order to reduce load on host machines, especially on the Linux architecture. We also plan to provide to the community a set of anonymized Internet traces with associated meta-data derived from gt in the near future, once we manage to clear the legal hurdles connected to such a release.

# 8. REFERENCES


[1] The Ground Truth software tools. http://www.ing.unibs.it/ntw/tools/gt.

[2] J. Erman, M. Arlitt, and A. Mahanti. Traffic classification using clustering algorithms. In *Proc. ACM SIGCOMM MINENET Workshop*, Pisa, Italy, Sep. 2006.

[3] H. Kim, K. Claffy, M. Fomenkova, D. Barman, and M. Faloutsos. Internet Traffic Classification Demystified: The Myths, Caveats and Best Practices. In *Proc. ACM CoNEXT*, Madrid, Spain, Dec. 2008.

[4] The Cooperative Association for Internet Data Analysis (CAIDA). http://www.caida.org.

[5] LBNL/ICSI Enterprise Tracing Project. http://www.icir.org/enterprise-tracing.

[6] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli. Tunnel hunter: Detecting application-layer tunnels with statistical fingerprinting. *Elsevier Computer Netw.*, 53(1):81–97, 2009.

[7] I. Trestian, S. Ranjan, A. Kuzmanovi, and A. Nucci. Unconstrained endpoint profiling (googling the Internet). *SIGCOMM Comput. Commun. Rev.*, 38(4):279–290, 2008.

[8] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. In *Proc. ACM SIGCOMM*, Philadelphia, PA, USA, Aug. 2005.

[9] G. Szabo, D. Orincsay, S. Malomsoky, and I. Szabo. On the Validation of Traffic Classification Algorithms. In *Proc. PAM2008*, Cleveland, OH, USA, Apr. 2008.

[10] M. Canini, W. Li, A. W. Moore, and R. Bolla. GTVS: Boosting the Collection of Application Traffic Ground Truth. In *Proc. 1st Intl. Workshop on Traffic Monitoring and Analysis*, Aachen, Germany, May 2009.

[11] L7 Filter. http://l7-filter.sourceforge.net.

[12] Tcpdump/Libpcap. http://www.tcpdump.org.

[13] M. Baldi, A. Baldini, N. Cascarano, and F. Risso. Service-based traffic classification: Principles and validation. In *Proc. IEEE 2009 Sarnoff Symposium*, Princeton, NJ, USA, Mar. 2009.

[14] P. Biondi and F. Desclaux. Silver Needle in the Skype. In *BlackHat Europe*, Amsterdam, The Netherlands, Mar. 2006.