Accurate Real-Time Labeling of Application Traffic

Sebastian Schäfer
schaefer@itsec.rwth-aachen.de
RWTH Aachen University

Alexander Löbel
loebel@itsec.rwth-aachen.de
RWTH Aachen University

Ulrike Meyer
meyer@itsec.rwth-aachen.de
RWTH Aachen University

***Abstract*—In this paper, we present the design and implementation of ATLAS, a novel tool for automatically labeling network packets with the process responsible for them. Our tool is able to label all kinds of outbound packets based on Windows events and TCP stream information with ground-truth accuracy. Additionally, it is able to label DNS packets with the correct process name instead of just the DNS resolver. Using ATLAS, it is possible to create large datasets, e.g., to create software fingerprints or train machine learning classifiers. Another use-case is to inspect the network traffic of a machine to determine which application is communicating with whom. We evaluate the performance considering different load scenarios to demonstrate the real-time capacity of ATLAS. Additionally, we analyze the communication endpoints of a Windows 10 host and compare the results before and after disabling all privacy related settings.**

I. Introduction

Many tasks in the area of network analysis and network security require the availability of large datasets. While one challenge is to collect such datasets, another challenge is to create labels with ground-truth accuracy. Especially for network-based application profiling or anomaly detection, the correct application or process that was responsible for each network packet is of interest. For example, to train a machine learning classifier for detection of a specific application or malware, correct labels of the training data are necessary. Also, it is generally not transparent how applications communicate via the network by simply monitoring the outgoing traffic of a host, e.g., whether an application transmits privacy critical activity data.

We present the design of Application Traffic Labeling Software (ATLAS), a tool for labeling outbound network packets on Windows with the application that produced them. ATLAS captures all outbound network traffic on a host and outputs a process-label for individual packets. This is done by parsing system logs to match individual system events to corresponding network packets to derive the responsible process and by keeping track of individual TCP connections for consistent labels. Additionally, ATLAS is able to assign correct labels of processes initiating DNS queries, instead of labeling all outgoing DNS traffic with the operating system's resolver. Our aim is to label packets with ground-truth accuracy. We evaluate the real-time capabilities of ATLAS in different load scenarios. Additionally, we demonstrate its capabilities of adding transparency to outbound network traffic, by comparing the network communication of Windows processes, with privacy and diagnostic settings enabled and disabled.

II. Related Work

Many approaches for network traffic classification [9], [13], [7], especially those based on machine learning, need datasets of application or malware traffic. While some approaches use unsupervised learning which doesn't require labeled training data, others rely on correctly labeled data which has many challenges [4]. This is especially important because ground-truth accuracy of labels has significant impact on the performance of classifiers, e.g., for malware detection [1].

Not much work was done when it comes to labeling individual packets with application or process names. In [14], the authors developed a tool that uses socket hooks and Network Driver Interface Specification (NIDS) hooks to get information about applications generating socket calls which is directly written to the packet headers. The tool does not seem to be publicly available and was developed for older versions of Windows. Because such socket hooks do not reveal information about applications initiating DNS lookups, we use system event logs for labeling. This allows us to label DNS queries with the initiating application instead of the operating systems resolver, which eventually sends the network packets.

Many existing approaches only apply labels to whole network traces instead of labeling individual packets, e.g., based on clustering. In [3], the authors use a semi-supervised labeling approach for network flows using clustering and self-training based on a few existing labels and apply this to the unlabeled data portion. In [2], the authors combine unsupervised and supervised learning for labeling network anomaly detection datasets by clustering the data, creating labels, and then training a model for labeling in a supervised fashion.

Additional approaches rely on existing detection systems to label network data after it was captured, which depends heavily on the accuracy of such systems. In [6], the authors retransmit captured data in a controlled environment and label it using an IDS like *Snort*. Similarly, in [8] the approach is to train a classifier for labeling based on the output of a variety of antivirus software combined with majority voting.

III. Design and Implementation

In the following, we describe the design of ATLAS and its components. We set the following requirements as design goals for ATLAS. Foremost, it should capture ground-truth labels with high confidence, i.e., it does not introduce false application labels. Then, it should have high coverage and not miss short-lived connections. Finally, we want to be able to label DNS requests not with the system's DNS resolver but with the application that requested the DNS lookup.

The design to achieve those goals is depicted in Figure 1. ATLAS captures all outgoing network traffic and stores it as a Pcap file, together with a CSV file containing the application labels for each packet. ATLAS is composed of three main threads. The event capture thread makes use of *wtrace* [10], a command-line tool that reports Windows events system-wide. We use the TCP and UDP handlers of *wtrace* to log and parse all reported network events. Each network event contains a timestamp, the application generating the event including its process ID and the source and destination IP addresses and ports. The parsed information is fed into the event queue for each event. Note that the events are reported in-order. Hence, the queue is always sorted by the event arrival time.

For the packet capture thread, we use the command-line network capture tool *tshark* [12] to record outbound network packets on a given interface. For each packet, we parse the timestamp, the frame number, the source and destination IP address and ports. This information is put into the packet queue which is again ordered, based on the packet timing.

To label the network packets with the application responsible for generating them, we use the information given by the event capture thread as follows. The label thread consumes events from the event queue and for each event, it searches for a matching network packet in a sliding window manner. This is realized by pre-selecting only those packets from the packet queue that have a timestamp t_{packet} in the interval $[t_{event}, t_{event} + x]$ where t_{event} denotes the timestamp of the current event and x is a user-configurable delay in seconds. The lower bound exists because packets are sent via the network interface only after the corresponding system-wide network event has occurred. The rationale behind the upper bound is to have plausible confidence in matching the correct packet to the event. Hence, the default value of x is set to 1 second to also take account for larger than usual delays.

For these pre-selected packets we consider only those where the source and destination IP addresses and ports are identical to the values of the current event. From these packets, we select the packet with the smallest difference between t_{packet} and t_{event}. If there is such a packet, the application name from the event is used as application label for this packet and is written to the CSV file. To re-identify the packet in the Pcap file of the collected traffic, the CSV file contains the corresponding frame numbers.

If there is no packet fulfilling these conditions, the event is dropped and the next one is considered. We argue that matching based on the IP addresses and port numbers together with a small allowed delay between Windows event and sent network packet leads to having high confidence in the accuracy of the labeling. However, this confidence decreases the more the allowed delay x is increased. If x is large, a different application might be re-using this IP and port combination, leading to falsely labeled packets. Our focus was to label outbound traffic accurately because inbound traffic is not entirely dependent on the locally installed applications, but on the external communication endpoint.

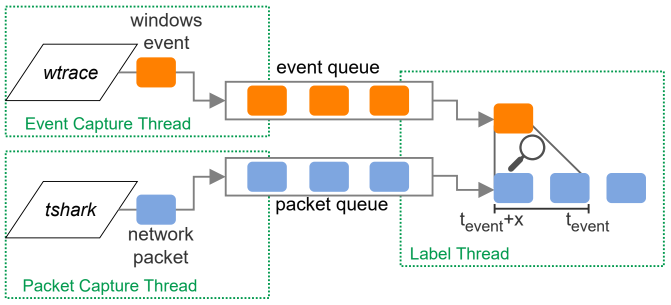

Fig. 1. Simplified design of ATLAS. The event capture thread logs Windows events and the packet capture thread captures information of outbound network packets. The label thread attempts to match each event to a packet.

In addition to labeling via event-packet matching, we make use of TCP streams for yet unlabeled packets that occurred earlier than the currently considered event. Once a packet receives a label through matching, we check if it has a TCP stream number. If so, we save the stream number together with the application name. Then, just before discarding the unlabeled packets earlier than the currently considered event, we check whether the packet has a stream number and if we already know an application label for the stream. If there is a stream label, we set the application of this packet to the same label as the other packets in the stream.

This allows us to label packets which we cannot correlate to an event, once we are sure that no future event can possibly match. Some packets, e.g., TCP keep-alive packets, do not trigger an event. However, by tracking TCP streams, we can still assign labels as long as we previously found a label for the same stream. To avoid having an ever-growing memory load, the storing of stream number and application pairs is realized through a ring buffer with configurable size.

A special case occurs when considering DNS requests. The application that is responsible for sending a DNS query is the DNS resolver, i.e., the Windows system process *svchost*. Hence, using the information from the reported event, all DNS requests would be assigned *svchost* as application label. However, it is more interesting to know about the application triggering the DNS request rather than the operating system's DNS resolver. Hence, in the case of DNS packets, we implement another matching mechanism with the help of Windows DNS logs as depicted in Figure 2. In the case of a DNS request with the *svchost* label, the label thread queries the Windows DNS logs with the domain and the timestamp of the DNS request. However, the matching DNS log entry contains only the process ID of the requesting application at that time. Hence, we need to match process ID and application name.

This is realized by another thread that continuously polls the mapping between application name and process IDs of the currently active processes with the help of the internal Windows tool *tasklist*. Every s seconds, *tasklist* is called and we save the process ID and the corresponding app name together with the timestamp when this was polled. The polling

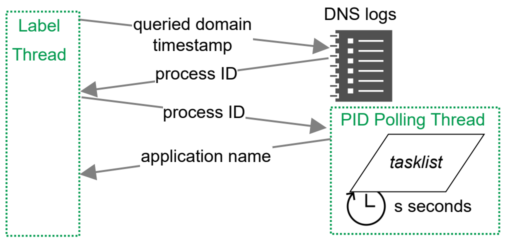

Fig. 2. DNS request labeling of ATLAS. The label thread looks up the process ID responsible for the DNS request, which is used to look up the corresponding application name using *tasklist*.

interval s can be set by the user and its default value is set to 0.5 seconds. We use a ring buffer with configurable size again to avoid unnecessary memory consumption. To get the application name, we choose the entry with the smallest timing difference to the DNS query timestamp – similar as with the event-packet matching earlier. If this is successful, the DNS request is labeled with the found application and stored into the CSV file. If we cannot find a matching application, e.g., because the polling frequency is too low, leading to missing short-lived processes, it is logged as "app name not found".

Lastly, we implemented a basic QT GUI which allows the user to choose the capturing interface, output paths for the CSV, Pcap, and logging files, as well as earlier mentioned values such as the maximum delay between events and packets. Furthermore, we include a statistics page showing the portion of currently labeled packets and the number of events and packets in their respective queues in real-time.

IV. Evaluation

In this section, we evaluate the performance of ATLAS and discuss its capabilities by measuring the impact of Windows 10 privacy settings on the network behavior. To measure the performance we used the Windows tool *perfmon* which allows capturing processor time as well as memory consumption based on private bytes and working set per running process. The processor time measures the time in percent that a logical core is executing a non-idle thread [11]. Hence, a value of 100% means that one core is under full load, while a value of 800% means that all cores of a 8-core CPU are under full load. The private bytes are those bytes that are allocated by the process without considering the memory that is shared with other processes [5]. The number of bytes given by the working set [5] is equivalent to the displayed memory usage in the task manager which equals the amount of pages the process is using in memory, excluding the ones paged out [5]. By that, we cannot measure the memory consumption exactly. However, both values represent a reasonable approximation.

Using *perfmon* with a sampling interval of 1 second, we captured those measures while running ATLAS in three different load scenarios on a Windows VM for an hour each. The idle scenario consisted of starting ATLAS and *perfmon* and not interacting further with the VM for one hour. The light load scenario consisted of web browsing to simulate typical office

TABLE I
Performance measurements of ATLAS in three load scenarios

	Processor Time [%]			Private bytes [MB]			Working set [MB]		
	min	avg	max	min	avg	max	min	avg	max
Idle load	0	2.5	100.1	215.4	271.6	317.5	283.3	337	388.6
Light load	0	23	129.7	217.5	327.2	513.3	285	444	583.5
High load	0	96.2	183	138	894.9	1643.1	165.8	964	1706.7

behavior. For the high load scenario, we initiated a *TeamViewer* remote control session where the VM was remotely controlled while transmitting its screen and joined a *Zoom* call including screen sharing and webcam transmission. All three scenarios were carried out on a Windows VM with 8 cores on an AMD Epyc 7702P CPU and 16 GB of main memory. The resulting packets per second (pps) were 0.36 pps (idle), 26.5 pps (light), and 262.1 pps (high).

Table I reports the minimum, average, and maximum of the processor time, the private bytes and the working set used by ATLAS in each of the three scenarios. Since ATLAS is composed of a Python program together with *wtrace* and *tshark*, we added up the values of each sub-process. In all scenarios, ATLAS managed to keep up with labeling the traffic, i.e., on average the event and packet queues were emptied faster than they were filled. In some instances, e.g., in the light load scenario when a new website was visited, it took multiple seconds to process all new events before catching up. This was achieved utilizing at most one core, with the exception of the maximum processor time in the browsing scenario where ATLAS utilized 130% processor time during a peak. However, in case of much higher traffic volume than the high load scenario, multi-core support for ATLAS would be beneficial, since the average CPU time was already at 97%, and the labeling of ATLAS is currently restricted to one core. The 183% peak CPU time was possible because, in addition to the single core for labeling, *tshark* and *wtrace* run using individual threads. In the high load scenario we measured a peak size of the working set of 1 707 MB. The average memory load was less than 1 000 MB with both metrics. We conclude that ATLAS is capable of labeling even high traffic loads in real-time, possibly with a small delay, while utilizing less than one CPU core on average.

We now describe a use-case for such labeled data, besides creating datasets for application profiling or malware detection. For this, we built statistics for each endpoint a host is communicating with by parsing the DNS requests and their responses which we captured in addition to outbound traffic for this experiment. As a result, we have a list of IPs and their corresponding domains that each application communicated with, including the amount of bytes sent to each IP or domain. We use those statistics to compare two settings. First, we set up a VM with a freshly installed Windows 10 and default settings and installed ATLAS. For the second setting, we cloned this machine and disabled all privacy critical settings. This included disabling ad tracking, location tracking, typing personalization, diagnostics data, background apps, and all

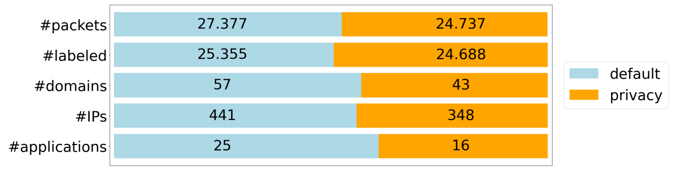

Fig. 3. Statistical data comparing the results of ATLAS for the default and privacy setting of Windows 10.

access to personal information from apps. On the first VM we used a Microsoft account, while we opted for a local account in the privacy focused setting.

The goal was to measure the difference in outgoing data when configuring Windows as restrictively as possible. To ensure that both VMs produce comparable data, we started both machines simultaneously without further interaction and left them running 24 hours. The collected data shows for each process which IPs it communicated with, the traffic volume, which domains it queried and which IPs correspond to each domain. In the following, we discuss whether the applied settings had any impact on the network behavior.

The results are depicted in Figure 3. Overall, the default VM sent more packets than the privacy VM. The system process *svchost* was responsible for 22 687 and 22 616 packets, respectively. Hence, the number of packets sent by other processes is more than twice as large in the default VM. Also, the privacy VM queried considerably less domains, communicated with less IPs, and and had less applications with network activity in general. On the privacy VM, 10 applications did not communicate compared to the default VM, namely *FileCoAuth*, *HxTsr*, *OneDriveStandaloneUpdater*, *RuntimeBroker*, *SearchApp*, *YourPhone*, *backgroundTaskHost*, *explorer*, *smartscreen*, and *tasklist*. For example, *tasklist* belongs to the task manager and is even used by ATLAS to match process IDs and names. On the default VM, 400 KB of data were sent to multiple IPs associated with Microsoft. *RuntimeBroker* is a Windows process that checks declared permissions of Windows apps, e.g., permissions to use the camera. The privacy VM not sending data was likely a direct result of restricting all applications not to use such resources. Other processes belong to services like *OneDrive*, *Outlook* and *Search*. Presumably, on the default VM these services have sent diagnostic data. However, because most traffic was encrypted, we were not able to verify this. Nevertheless, the IPs and corresponding domains the applications communicated with give more information about their potential use. E.g., *activity.windows.com* was accessed by *svchost* only on the default VM and can be associated with activity tracking.

V. Conclusion and Future Work

In this paper, we presented the implementation and evaluation of ATLAS, a tool for Windows that is able to label network packets with application names in real-time with ground-truth accuracy. The tool will become open source alongside the publication of this paper. ATLAS makes use of different system event logs to assign the labels, including precise application labels for DNS queries instead of the OS resolver. Additionally, packets that cannot be associated with an event can still be labeled based on TCP stream information. We evaluated the performance based on three load scenarios and showed that ATLAS manages to label high traffic volume while utilizing less than one CPU core and less than 1000 MB of memory on average. Furthermore, we demonstrate its capabilities by analyzing the impact of Windows privacy settings on network connections. For future work, we envision support for Linux and the addition of multi-core support to allow for real-time labeling of even higher traffic volume. Also, we want to add version information to the labels and group the process labels into an application label based on paths of the process executables. Additionally, we plan on correlating inbound traffic to specific processes.

VI. Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 833418. Alexander Löbel was supported by the research project "North-Rhine Westphalian Experts in Research on Digitalization (NERD II)", sponsored by the state of North Rhine-Westphalia – NERD II 005-2201-0014.

References

[1] B. Anderson and D. McGrew, "Machine Learning for Encrypted Malware Traffic Classification: Accounting for Noisy Labels and Non-Stationarity," in *Conference on Knowledge Discovery and Data Mining*. ACM, 2017.

[2] S. Baek, D. Kwon, J. Kim, S. C. Suh, H. Kim, and I. Kim, "Unsupervised Labeling for Supervised Anomaly Detection in Enterprise and Cloud Networks," in *4th International Conference on Cyber Security and Cloud Computing*. IEEE, 2017.

[3] A. Fahad, A. Almalawi, Z. Tari, K. Alharthi, F. S. Al Qahtani, and M. Cheriet, "SemTra: A Semi-Supervised Approach to Traffic Flow Labeling with Minimal Human Effort," *Pattern Recognition*, vol. 91, pp. 1–12, 2019.

[4] J. Guerra, C. Catania, and E. Veas, "Datasets are not Enough: Challenges in Labeling Network Traffic," *Computers & Security*, p. 102810, 2022.

[5] R. Mariani. (2004) Performance Planning. Accessed: 08.04.2022. [Online]. Available: https://docs.microsoft.com/en-us/archive/blogs/ricom/performance-planning

[6] K. Masumi, C. Han, T. Ban, and T. Takahashi, "Towards Efficient Labeling of Network Incident Datasets Using Tcpreplay and Snort," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021.

[7] A. W. Moore and K. Papagiannaki, "Toward the Accurate Identification of Network Applications," in *Passive and Active Network Measurement*. Springer, 2005.

[8] S. Nari and A. A. Ghorbani, "Automated Malware Classification based on Network Behavior," in *International Conference on Computing, Networking and Communications*. IEEE, 2013.

[9] T. T. Nguyen and G. Armitage, "A Survey of Techniques for Internet Traffic Classification using Machine Learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[10] S. Solnica, *wtrace*, https://wtrace.net/documentation/wtrace/.

[11] H. Tarra. (2012) Understanding Processor (% Processor Time) and Process (%Processor Time). Accessed: 08.04.2022. [Online]. Available: https://social.technet.microsoft.com/wiki/contents/articles/12984.understanding-processor-processor-time-and-process-processor-time.aspx

[12] *tshark(1) Manual Page*, https://www.wireshark.org/docs/man-pages/tshark.html.

[13] S. Zander, T. Nguyen, and G. Armitage, "Automated Traffic Classification and Application Identification using Machine Learning," in *Conference on Local Computer Networks 30th Anniversary*. IEEE, 2005.

[14] C. Zhao, L. Peng, B. Yang, and Z. Chen, "Labeling the Network Traffic with Accurate Application Information," in *8th International Conference on Wireless Communications, Networking and Mobile Computing*. IEEE, 2012.