Automated Discovery of User Trackers

Sakshi Jain

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-229
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-229.html

December 19, 2014

Copyright © 2014, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to take this opportunity to thank my advisor Prof. David Wagner, you have been a great mentor to me. I would like to thank you for constantly encouraging me exploring my interests in security and my research. Your advice on both research and my career has been immensely helpful. I am grateful to Prof. Vern Paxson for guiding me very closely for my thesis project and teaching me how to do diligent and methodical research. Thank you Mobin Javed for long discussions over tea about research problems, career and life in general. Thank you for your contribution in analysis section of the project and for spending time directing the project. My research would not have been possible without the generous support of ISTC for Secure Computing, Intel.

Automated Discovery of User Trackers

by Sakshi R. Jain

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Vern Paxson
Research Advisor

(Date)

* * * * * * *

Professor David Wagner
Research Advisor

(Date)

Acknowledgment

I would like to take this opportunity to thank my advisor Prof. David Wagner, you have been a great mentor to me. I would like to thank you for constantly encouraging me exploring my interests in security and my research. Your advice on both research and my career has been immensely helpful. I am grateful to Prof. Vern Paxson for guiding me very closely for my thesis project and teaching me how to do diligent and methodical research. Thank you Mobin Javed for long discussions over tea about research problems, career and life in general. Thank you for your contribution in analysis section of the project and for spending time directing the project. I would also like to thank my friends and family who have been my support during all the ups and downs of a grad life. My research would not have been possible without the generous support of ISTC for Secure Computing, Intel.

Abstract

Web tracking, the practice by which web sites collect information about the user's browsing history across one or more sites, is highly prevalent on the web today. This is done using unique identifiers (*trackers*) that can be mapped to client machines and user accounts. Although such tracking has desirable properties like personalization and website analytics, it raises serious concerns about online user privacy. Conventional trackers like browser cookies and Flash cookies are widely known to the community; however there is potentially more tracking information being sent to servers around the world unbeknownst to the users and security community at large.

This work is motivated by the possibility of discovering previously unrecognized forms of trackers, either potential or actual in an "automated" fashion from raw network traffic. In this work, we built a tool that processes users' network traces and outputs tracker strings such as usernames, cookies, IMEI numbers and the like, that uniquely identify a machine/device/browser. The key challenge in automatically capturing trackers from raw traces is dealing with enterprise-sized data. We tackle this problem by applying data-driven multi-stage filtering, thereby pruning the size of network traces to be analyzed. Each filtering step has a trade-off between between false positive rate and potentially interesting information lost (false negatives). It is important to carefully maintain a balance between the two while choosing each new filter. Our tool uses six major filters and outputs a set of potential trackers for each user in the network. We found trackers that were sent as a part of URL parameters, User Agent, as well as in the non-HTTP payload apart from cookies.

1 Introduction

The term *web tracking* has been traditionally used to describe the practice by which websites collect information about a user's browsing activity across one or more sites. In this context, a *tracker* is any piece of information that can *uniquely* identify web activity by a user. Some commonly known trackers include first and third party cookies, Google Analytics cookies, Flash cookies and usernames. Some trackers do not identify the human user, rather identify the client machine or browser instance. For example, username can be used to track a user across various client machines, vs some third-party cookies can only track a browser instance. How long can a user be tracked using the same tracker string depends on the server that is tracking and the purpose of tracking. For example, *session cookies* are trackers that last for a browsing session and are deleted when the user closes the browser, whereas *persistent cookies* often last for years and get deleted only when they expire or the user explicitly deletes them. Our work is motivated by the possibility of discovering previously unknown forms of tracking mechanisms, either potential or actual.

Identifying new potential trackers can help us answer two questions. First, what are the different classes of web-tracking? Do we find any examples of *tracking mechanisms* not known to the community? Second, how much tracking power can an organization with eavesdropping power achieve? The higher the frequency with which these *trackers* appear in a network, and the greater the number of networks the adversary can eavesdrop, the higher tracking power it achieves. The implications of this tracking power are two-fold: (*i*) an adversary can identify the presence of a machine on a network by building a rich set of trackers particular to a machine, and (*ii*) a site or an enterprise can attribute individual network flows in NAT'ed traffic to the respective users behind the NAT.

Previous works on host and user tracking have manually identified potential tracker strings and studied their effectiveness in uniquely identifying a host or a user. Third party cookies, User Agent, username and IP address are some examples of such previous identified trackers. Our work, on the other hand, examines whether we can automatically capture previously unrecognized forms of trackers, either potential or actual, from raw network traces. In an effort towards this direction, we built a tool that achieves this semi-automatically. The tool uses no knowledge of application layer protocols and works directly on the payloads at transport layer. It processes users' network traffic and outputs candidate tracker strings like cookies, tracking parameters in URLs, user IDs, Google Analytics cookies and the like. It is largely automatic except for a manual analysis component as the last step to validate the output of the tool.

The working of the tool depends on characterizing *potential* trackers by repeated occurrence of a string, seen only from one machine/device in the network. Note that we use *potential* because this definition does not exclusively characterize trackers. It is possible that this network-view approach classifies some of the actual trackers as non-trackers, because of two reasons: (i) the string did not repeat, i.e., was only observed once for a user in the observance window, or (ii) the tracker is user-specific rather than device-specific, and hence repeats across multiple devices (belonging to the same user). Further, in this work we are interested in finding trackers that are persistent. Ideally, the longer a tracker continues to uniquely identify a machine, the stronger is its tracking power. Since size of network traces increases quickly with the size of the enterprise, dealing with the size of data and validating true trackers while weeding out false positives are the key challenges in this work. Our tool makes use of two key ideas along with various optimization techniques to tackle these problems, namely, (*i*) multistage filtering and (*ii*) streaming algorithms. We found that, though the tool uses very generic techniques that are not application dependent, it was able to efficiently capture quite a few novel tracking strings

apart from the well-known trackers like cookies. We believe that the design of our tool is such that it is memory efficient and can handle reasonably large size of input traffic.

The remainder of this thesis is organized as follows. We begin with related work. Sec. 3 details the characteristics of the dataset we use in building and evaluating the tool. Sec. 4 outlines a naive approach to detecting tracker strings followed by challenges associated. We motivate and briefly discuss the techniques used in our tool in this section. Sec. 5 describes the various components of our tool and Sec. 6 enumerates the multistage filtering technique and the various filters employed. We describe detailed implementation of our tool in Sec 7. Finally we discuss the analysis results in Sec. 8 and conclude with Sec. 9.

2 Related Work

Literature relevant to our work can be divided into two domains, (*i*) privacy diffusion and (*ii*) methodology for pattern recognition on network traffic.

Leakage of both private information about a user's browsing activity across various websites and personally identifiable information[1] has attracted attention by security and privacy researchers. Krishnamurthy et.al. [12] highlight the seriousness of the privacy diffusion problem by examining the penetration of identifiers placed by third party aggregators in overall web traffic across 4 years. They show that the penetration of top 10 third-party servers tracking user-viewing habits across a large set of popular websites grew from 40% in Oct'05 to 70% in Sep'08.

Many past efforts focused on capturing identifiers in network traffic that can be employed for host tracking or obtaining PII unbeknownst of the user. It has been shown that third-parties can link PII leaked by Online Social Networks (OSN) with user activities both within and outside the OSN sites [11]. They achieve this by studying the leakage of unique identifiers like usernames used by OSNs as a part of Referer field, request URI, and Cookie fields in the HTTP requests sent to third parties. Previous studies on host tracking, also called host fingerprinting, leverage packet-level information to identify the differences in software systems [3, 4, 5] or hardware devices [9]. Other works on tracking web clients require probing hosts' system configurations [8], the installation order of browser plug-ins [13] or application level IDs [15]. [16] compares the effectiveness of host identifying information revealed by a variety of trackers like User Agent, IP prefix, Cookie ID, and its implications on cookie churning. They show that 88% of returning users can still be tracked even if they clear cookies or utilize private browsing. All of the previous works on leakage of user-identifiable information depend on first manually identifying pieces of interesting information (*trackers*) sent by user's machine on the network and then using one or combinations of these pieces for tracking. Our work is comparatively different since it focuses on automatically discovering previously unknown forms of tracking identifiers from raw network traces.

The methodology used by our tool consists of both existing and novel techniques for mining interesting contents from network traffic. Sumeet Singh et. al. built EarlyBird, a prototype for automated extraction of worm signatures from the network [14]. They do this by characterizing worms as strings sent in the network that are highly prevalent and widely dispersed. They develop multiple counting algorithms to sift through network contents for such strings they characterize as worms. Honeycomb is another host-based intrusion detection system employs

[1]Personally Identifiable Information is (PII) any information that can be used to distinguish or trace an individual's identity either alone or when combined with other information that is linkable to a specific individual [11]

Table 1: Statistics about data used

Count of unique IPs	512
Total number of days	15
Count of NAT IPs	4
Count of unique MAC addresses behind NAT	290
Count of users outside NAT	500
Total count of users	790
Count of users with non-zero contents	786

pattern matching to identify NIDS signatures as longest common sub-sequence from strings found in message exchanges with honeypots [10]. Like our tool, neither of these two systems use knowledge specific to the application-layer protocol. We use a similar underlying method to characterize what *trackers* are, sift through network traffic and apply multistage filtering algorithms to automate their detection.

3 Data

In this section, we describe the dataset used in this report. For our analysis, we use fifteen days' raw network traces captured at the border router of ICSI's (International Computer Science Institute) network. This network contains 512 unique IP addresses in the address range 192.150.186.0/15 with 4 IP addresses corresponding to NATs. We use the DHCP and NAT logs to resolve individual machines behind the four NATs. DHCP logs provide a mapping between MAC address and internal IP while NAT logs provide a mapping between private connection tuple and public connection tuple. Using these two logs, one can map public source IP and public source port to a MAC address behind the NAT for a given timestamp. The ICSI network has in total of 290 unique MAC addresses behind the 4 NATs. We have about 790 (500 non-NAT + 290 behind NAT) distinct "users" in our dataset where 786 users have non-zero traffic content. For the sake of simplicity, we consider each IP and each unique MAC address behind NATs as a separate user for our analysis in the remaining part of the report even though the same individual user can own a desktop with a fixed public IP and a laptop connected to a NAT, thereby occupying two users instead of one in our users list. The total size of out dataset is approximately 3.5 terabytes with an average size of 4.4 gigabytes per user and 274 megabytes per user per day.

4 Key Challenges

From the definition of trackers, we conclude that we can characterize all candidate user-identifiable strings as strings occurring repeatedly on a network and seen from only one device/machine. There are multiple naive approaches that can capture strings that satisfy these two requirements. In this section, we show one possible algorithm and discuss the key challenges with this naive approach thereby motivating the need to develop other sophisticated techniques. We end this section by giving an overview of the two key techniques used in our tool to solve this problem.

Algorithm 1: Naive approach to uniqueness

Input: $user_1, user_2, ...user_n$, where $user_i$ is a list of strings for user i
Output: list of unique strings for each user

```
1  INITIALIZE string_count to dictionary of form {str: list of users}
2  for each user i do
3  |  for each string str in user_i do
4  |  |  ADD user_i to the list string_count[str]
5  |  end
6  end
7  for each str in string_count do
8  |  DELETE str if count of users for str > 1
9  end
10 for each user i do
11 |  OUTPUT user_i ∩ strings in string_count
12 end
```

4.1 Naive Approach

We present one possible approach to obtaining unique persistent strings in the network. Algorithm 1 provides pseudo code for obtaining a set of unique strings for each user in the network. The algorithm goes as follows. For each user i, separate out the traffic contents from the network and extract substrings of all possible lengths into a list $user_i$. Initialize a global table $string_count$ which maintains a list of unique users each substring occurred for. Scan through the list of substrings for each user and populate the entry in table $string_count$ appropriately. Once the contents of all the users have been processed, delete the substrings from the table for which the count of unique users was greater than one. We are now left with a global list of just those substrings for which there was exactly one user it was found in. In order to map these substrings to the desired user, we output the intersection of the respective user's strings with the strings left in the table $string_count$. This step ensures the uniqueness property per user.

In order to now restrict the substrings per user to those which were persistent across a certain time window, use the same naive approach except now, instead of maintaining a global dictionary of strings and their associated list of users, maintain a dictionary of strings and a list of days it occurred on. Once the persistence check has been applied too, we are left with desired tracker strings for each user.

4.2 Challenges

The approach described above clearly achieves our objective of weeding out any user content which was either not unique or was not persistent across a certain time window. However, in its current form, Algorithm 1 is far from practical both in terms of both time and space requirements. We go over the key challenges posed by this approach as follows.

Scale

In the current form, the memory requirement for Algorithm 1 is strictly greater than the total size of all unique substrings across all users because of the need to store the table $string_count$ in memory while processing. Algorithm 1 extracts out substrings of all possible lengths from each user's network traffic. This is ideal since it would give us exactly the user content that is

interesting. Note however, that the memory requirement for storing all possible substrings is of the order O($\sum_i L_i^2$) where L_i is the total number of bytes in the network trace of user i. For the dataset we are working with, the total size of input trace files is about 3.5 TB, and the naive algorithm would need far more memory than that. Since the resources available at hand cannot process this big data in memory, it becomes important to explore other techniques for attacking the problem.

Validation

Validating whether a string output by Algorithm 1 is truly a tracker is another key challenge in using this approach. The number of false positives (substrings which are not trackers but marked as trackers by the algorithm) obtained by simply applying the conditions of uniqueness and persistence can be large. Since we manually perform validation of candidate substrings output by the tool, a high false positive rate leads to a large burden on the analysis. To intuitively understand why the number of false positives would be large, consider the following scenarios.

1. *Server connection unique to a user:* If a user regularly establishes connections with a server S to which no other user in the network connects, with high probability any content sent to this server would be unique to the user and our algorithm would wrongly mark it as a tracker.

2. *Server content unique to a user:* Consider a scenario in which a user connects to a common server but repeatedly visits a unique webpage, for example, a song on some radio station, or a video on youtube.com. Any content from this web page would be unique to the user and our algorithm would wrongly mark it as a tracker.

3. *Encrypted traffic:* Our algorithm would wrongly mark a large proportion of encrypted data as trackers. This is because repeating cipher text marked as trackers would not imply repeating underlying plain text. In fact, cipher text found repeating by the algorithm would most likely be due to collision of the outputs of the encrypting function for different inputs.

4.3 Key ideas

The challenges mentioned above make the problem of identifying user trackers in network traffic a difficult one. In order to tackle these hurdles, we propose a two-fold approach.

Multistage Filtering

The key objective of this work is not to be comprehensive in identifying *all* tracker strings but to capture as many new trackers as possible. This relaxed goal allows us to first remove any content which has a low probability of finding trackers and then apply the constraints of uniqueness and persistence on a smaller set of remaining data. For our tool, we employ multiple filters to achieve this. For each filter, we consider the trade-off between the processing benefit and reduced false positive rate obtained by using the filter on one hand, and true interesting trackers lost on the other. This approach of progressively winnowing down the input stream not only makes our tool more scalable but also reduces the overhead of manual analysis for validation.

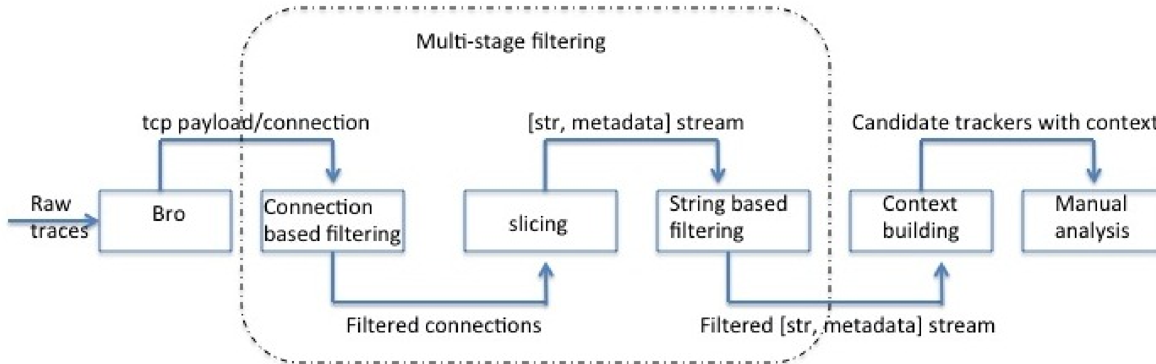

Figure 1: How we process network traces to identify trackers

Streaming Algorithms

Though multistage filtering provides a large reduction in the size of data to be processed, directly applying Algorithm 1 is still not feasible due to the sheer size of the input. To deal with this situation, we develop streaming algorithms that can accurately and efficiently perform filtering with more modest memory requirements. The key idea is to use streaming which processes data streams in which input is presented as a sequence of items and can be examined in only a few passes. These algorithms use memory much smaller than the total size of input and are typically used when the entire dataset cannot fit in memory. We would however like to point out to the reader that our algorithm is not entirely streaming. We do rely on a sorting step performed on each user's partial network traffic which does have non trivial memory requirements.

5 Architecture Overview

In this section, we give a step-by-step overview of our approach. We decompose the processing task into a pipeline of filters. Figure 1 shows the entire pipeline schematically. We defer the details of the implementation including the rationale behind each component until the following two sections.

We process raw network traces captured at a border router. We preprocess these traces using network traffic analysis tool Bro [2] to produce content files for each user. Each content file corresponds to the stream of TCP payload data for a separate TCP connection. The first filtering step, *connection-based filtering*, uses the metadata associated with each connection, i.e., information contained in the five tuple ⟨P, sourceIP, sourcePort, destinationIP, destinationPort⟩, where P is the transport layer protocol, to either filter it out or feed it to the next step. Of the connections that remain, we slice the traces into sliding windows of k bytes which we refer as "strings". To be able to rebuild the original context from the sliding windows for analysis, we associate each string with some metadata. This metadata includes the time window in which the string appeared, the path to the content file it appeared in, and the byte offset in the corresponding content file. By the end of this step, we have a list of k-byte strings and associated metadata for each user. Before feeding these to the next step, we sort the list of strings for each

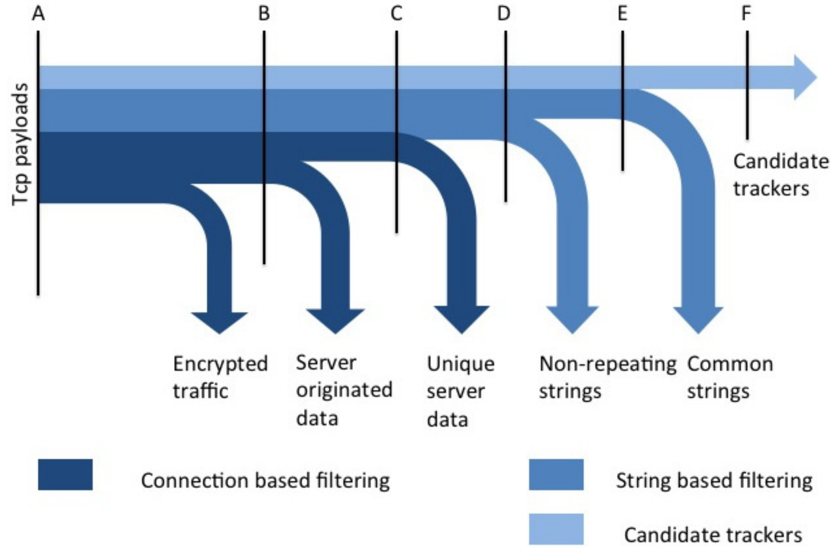

Figure 2: Various filters in order as they appear in our tool

Table 2: Filtering strengths of all connection-based filters

Filtering Stage	Filter Name	Inter-quartile range	Median
Connection-based	Encrypted traffic	314.5	1.0
Connection-based	Server Originated traffic	1.3	2.0
Connection-based	Unique Server data	160.0	1.2

user. The next step, *string-based filtering*, filters these strings using various uniqueness and persistence conditions. The output of these steps is a small set of candidate tracker strings for each user. We analyze the candidates manually for validation. To assist manual analysis, we rebuild the original strings from their sliding windows. This helps us in the validation process by providing the context in which the string appeared in the original trace, for example, whether it was a part of the URL or Cookie or Referer or in the payload. The final output of our tool is the set of strings that we believe can be used as trackers and are worth highlighting to the security community at large.

6 Multistage Filtering

In this section, we describe the various filters contained in connection and string based filtering steps. A schematic diagram with the various filters is shown in Figure 2. For each filter, we discuss in detail what the filter achieves and the trade-offs considered while selecting it.

6.1 Connection-based filtering

This step performs filtering based on the properties of a connection, identified by the five-tuple ⟨P, sourceIP, sourcePort, destinationIP, destinationPort⟩, where P is the transport layer protocol. We remove three kinds of content from our analysis: (*i*) encrypted connections, (*ii*) connections to servers visited by only a single user, and (*iii*) all data sent by the server to the client, for all

connections. In this section, we discuss the trade-of for choosing each filter, how we implement the filter, and its *filtering strength*. We define *filtering strength* for a user as the ratio of filtered TCP payload and total size of input TCP payload. Note that our input data contains connections made from outside ICSI to within ICSI as well as machines which are dormant. Since the data is skewed w.r.t. size of network traffic on each machine, we describe the overall filtering strength using median and interquartile range of individual user filtering strengths. Note that each filter's strength depends on which filters ran previous to it; for uniformity, we quote the *standalone filtering strength* of each filter. If k the input to the tool, standalone filtering strength is the value obtained by dividing the size of k by the size of the output when the filter under consideration is applied directly on k.

Remove encrypted connections

We identify encrypted connections as those where the destination port is in the list {22 (SSH), 443 (HTTPS)}, and remove both the originator and responder bytestreams of such connections. This filter is motivated both by the reduction in false positives and the difficulty in validating trackers manually. Repeating cipher text caught by our tool does not correspond to repeating underlying plain text and typically does not correspond to a tracker. Validation too is difficult since the strings captured are not human readable and we cannot obtain the underlying context in which they were sent. Note that by choosing this filter, not only do we let go of trackers sent in encrypted form but also some of the potentially interesting information that was sent as a part of the encryption protocol in plain text, for example, TLS handshakes.

The overall filtering strength of encryption-based filter is given by the interquartile range of about 314.5.

Remove server-originated bytestream

Next, we remove all responder bytestreams, i.e., all data sent by a server to a client. Trackers have to appear in client-originated data bytestreams, for a server to use them to identify a client. Typically, they will occur atleast once from the server and many times from the client. Therefore, it suffices to look at just one of the two directions. We focus on client-originated data and ignore server-originated for three reasons: (*i*) there is less client data typically, (*ii*) trackers typically occur more in client data and hence are easier to detect and (*iii*) server-originated content can lead to many false positives. This is because it is very common to have content such as web pages, videos or documents repeating for the same user, but not across users. For example, consider a user listening to the a song on multiple days, while no one else on the network listens to the same song by the same provider. This would cause a false positive (it would look like a tracker) if we didn't filter out server-originated data. Thus, we largely reduce the number of false positives by removing the server-side content. We do, however, lose some potentially interesting strings. For example, consider a site which shows a cleartext welcome page with the user's unique username post authentication. However, the proportion of such potential trackers is very small as compared to the huge increase in computation that would be needed to find them.

The overall filtering strength of server-based filter is given by the interquartile range of about 1.34. The value is much lower than expected since there are multiple connections made to servers within ICSI from outside and these servers act as individual users in our dataset. The average of individual user filtering strength is however 24.7 and maximum value attained is 4674.8.

Table 3: Filtering strengths of all string and context baed filters

Filtering Stage	Filter Name	Average
String-based	Non persistent strings	1.5
String-based	Common strings	26.7
Fine-grained filtering	Context Filtering	1.7

Remove unique server connections

If only a single user in the network connects to a particular server, all of the connections between the user and the server are filtered out in this step. Unlike the previous two filters, which can be independently applied to each user, this step not only depends on the connections of the user under consideration but also on the connections made by other users in the network. For simplicity, we identify each server by its IP address rather than its domain name. It is possible that the same domain is accessed by two users with different server IP addresses and hence marked as unique for each, thereby wrongly filtering out all the content from this domain for both the users.

A major benefit of using this filter comes from the reduction in false positive rate. Any content unique to the server would be almost surely wrongly marked by the tool as a candidate tracker for the corresponding user. This would include server-specific URLs, web pages and documents which in reality do not serve as good trackers. Similar to the previous filter, we do potentially lose a few trackers.

The filtering strength of this filter largely depends on the connections made by other users in the network. The overall filtering strength of this filter is given by the interquartile range of about 160.

6.2 Slicing

This step creates a sliding window of strings from the network traces. We choose 8 bytes as the size of the sliding window based on two factors, (*i*) processing overhead and (*ii*) interesting tracking strings lost. The total size of substrings obtained using a window of size k bytes would be k times the input trace file. Thus, we would like k to be as small as possible. However, also note that any tracker string with length greater than k could be lost in the process since it is not necessary for a substring of a tracker to remain unique and hence the tool might filter the substring out. The plot in Figure 3 shows a CDF of the count of unique strings (hence potential tracker strings) vs. their frequency for the values of k in the set $\{8, 64, 128, 256\}$. Since the number of unique strings does not change much going from $k = 256$ to $k = 8$ while giving a large reduction in total data size, we choose $k = 8$ as the size of our sliding window.

6.3 String-based filtering

In this step, we perform filtering based on the properties of individual strings in user traces. Recall that the input to this step is a list of eight-byte strings and associated metadata per user, generated using a sliding window on the connections left after the *connection-based filtering* step. We apply two filters in this step: (*i*) remove non-persistent strings, and (*ii*) remove strings common across users, corresponding to the two conditions of persistence and uniqueness as required by the definition of trackers. For each filter, we provide *filtering strength* described by

the total size of out of the filter across all users divided by the total size of the input to the filter across all users.

Remove non-persistent strings

This step weeds out all strings which do not repeat over multiple time windows, where the time window can be as small as an hour to as big as a week. The bigger the time window, the stronger is the tracker in terms of its tracking power. In our analysis, we use a day as the time window. Thus, our tool only retains strings which are observed on two different days. This filtering step can be independently applied on each user's list of strings.

Note that any string that occurs only once in a user's trace is automatically filtered out in this step. This might mean losing some true trackers which happen to appear only once in our dataset of limited size. However we accept this trade-off since without multiple occurrences we cannot validate if the string is a tracker. Thus, this filter not only helps us in strengthening the trackers found but also in reducing the false positive rate.

The filtering strength of this step is about 1.5.

Remove strings common across users

In this step, we filter out any strings that are observed across multiple users. This filter follows by our definition of a tracker that a tracker must be unique to a user. Note that in this step, we also filter out strings which are true trackers but appear in traces from two different *users* due to a single individual using multiple devices. For example, we would filter out a username on a site that is accessed by a user on their phone as well as laptop. However, if there exists at least one user in the network who accesses the same site only using a single device, our tool will capture the tracker. Thus, in this step, we might lose a few user-specific trackers but we do not lose any device-specific trackers.

The filtering strength obtained in this step is approximately 26.7.

7 Implementation

In this section, we go over the details of implementation of the various components of the tool along with the tricks and techniques used to reduce its memory footprint. As a reference, a schematic visualization of the tool and the various filtering steps are shown in Figure 1 and Figure 2 respectively.

We use the network traffic analysis tool Bro [2] to extract the TCP payload per connection for each user in content files. We perform the next step, i.e., *connection based filtering*, by imposing checks on the five-tuple ⟨P, sourceIP, sourcePort, destinationIP, destinationPort⟩, where P is the transport layer protocol. The connections which are not filtered out are sliced using a sliding window of size 8 bytes to generate a stream of strings and associated metadata for each user. Each element in a user's list of strings looks like ⟨ *str*, D, P, B ⟩, where *str* is the value of the string, D is the day it occurred on, P is the path to corresponding content file and B is the byte offset. By choosing a sliding window instead of all possible substrings, we reduce the total size of substrings from quadratic to linear in the size of the input trace. In order to be able to rebuild the original context from the sliced strings, we use the byte offset and path to the original content file stored in the metadata associated with each string. Thus, we not only reduce the total size of data to be processed, but also do not lose any information.

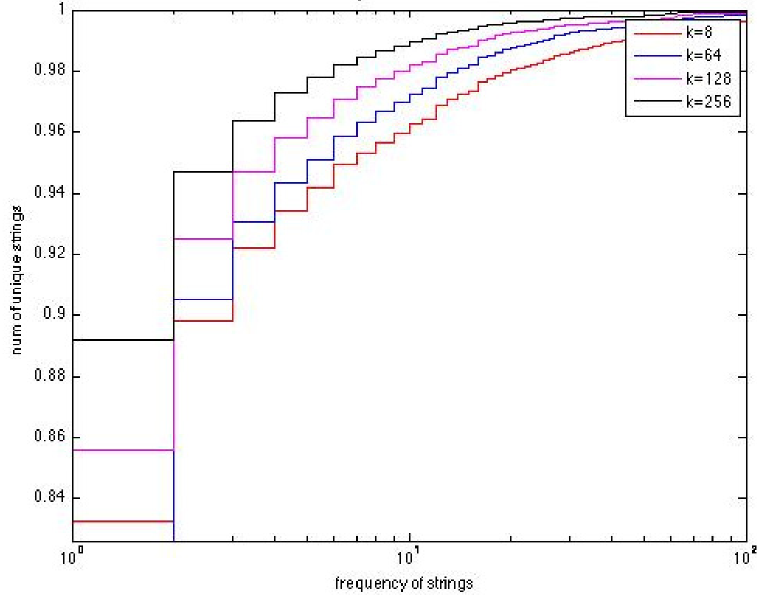

Figure 3: log-linear plot of cummulative distribution function of count of unique strings vs. their frequency of occurrence in one user's trace

Before proceeding to the next step of sorting, we place the strings into 256 different buckets depending on the ASCII encoding of their first character, for example, all strings starting with "a" will go to bucket number 65. Since in string-based filtering, whether a string is filtered out or pushed to the analysis stage solely depends on the various statistics corresponding to the particular string, bucketing does not interfere with the processing but in fact partitions the data nicely to make sorting easier. The output at this stage is a folder structure with 256 buckets each containing n files, n being the number of users. Each file contains all strings of that user starting with a particular character. Sorting is the only non-streaming component in the tool. The memory footprint of the tool mainly depends on how efficiently we can sort each of these files. Note that, since sorting is performed on each user and each bucket separately, the tool will remain tractable as the number of users increases, the bottle-neck being the user with biggest trace size.

The next step, *string-based filtering*, ensures uniqueness and persistence of the strings left. Though the size of input to this step is much smaller than the initial size, it is still not small enough for the naive Algorithm 1 to work efficiently. In order to surmount this hurdle, we resort to a specific class of algorithms called *streaming algorithms*. Streaming algorithms process massive data like the network traces in our case and assume that the data is presented as a stream. Since storing and indexing of large amounts of data is expensive and may not always be feasible, this technique processes data as it appears and uses limited processing space (less than linear in size of the input). Such algorithms make very few passes over the data, typically just one, but at no point is the entire dataset loaded in memory. We develop filtering algorithms using this technique for the two string-based filtering steps, namely removing non-persistent strings and removing common strings. Note that though using streaming algorithms help us in reducing the memory consumed by the tool, the disk storage space required for subsequent processing steps increases.

7.1 Removing non-persistent strings

This filter weeds out any string in a user's trace that did not repeat across multiple days. The input to this step is a list of strings for each user, and associated with each string is the day it occurs on. A pseudo code for the streaming algorithm for this filter is provided in Algorithm 2. Input data is read as a stream of objects where each object is a [*str, day*] tuple. Since the input is sorted according to the string, information about all the days the string occurred on appears adjacent to each other in the stream, making it possible to process the data in a streaming fashion. In order to decide whether a string under consideration should be filtered or not, the algorithm maintains the information about the string value ($curr_str$), day ($curr_day$) and its start line ($start_line$) from the first time the string appeared in the stream. As the stream proceeds, either the same string would reappear with same or different days or a new string would appear. If before the new string appears, no day other than $curr_day$ was observed for string $curr_str$, we know that the $curr_str$ did not appear on multiple days and hence the algorithm filters it out. If on the other hand, a different day ($\neq curr_day$) was observed for $curr_str$ before seeing a new string, it implies that the string appears on multiple days and hence the algorithm prints it to the output filtered stream.

At any point in time, Algorithm 2 maintains only a constant number of variables and 2 pointers to the streams in the memory. Since the memory consumption is independent of the size of the input stream, this algorithm is easy to scale for much larger data sets.

7.2 Removing common strings

Pseudo code for the streaming algorithm for identifying unique strings for each user is provided in Algorithm 3. The input to this step is n streams $\mathbf{u^1}, \mathbf{u^2}, ..., \mathbf{u^n}$, where n is the number of users and each stream $\mathbf{u^i}$ is the output of the previous persistence filtering step for user i. Since information about the day associated with each string is not relevant for this step, we represent an object of a stream simply by the string value str. Note that similar to the previous case, each input stream $\mathbf{u^i}$ to this step is sorted on the ASCII encoding of str. The algorithm maintains a list of pointers $curr_strs$ to the current str in each stream $\mathbf{u^i}$. It also maintains a separate list of pointers to the output stream of each user $\mathbf{o^1}, \mathbf{o^2}, ..., \mathbf{o^n}$ where it prints all the filtered content. At every step, the algorithm generates a list of streams $min_streams$ which point to the smallest string (comparison is performed based on ASCII encoding) in $curr_strs$. If this list contains more than one stream, the algorithm filters out this string from each stream it was observed in and moves on to the next string. If on the other hand, the $min_streams$ list contains only a single stream, say $\mathbf{u^i}$, it means that the smallest string was unique to the stream $\mathbf{u^i}$. All the occurrences of this string in $\mathbf{u^i}$ are then copied over to $\mathbf{o^i}$ and the next string in $\mathbf{u^i}$ is loaded into $curr_strs$. This goes on, until all streams have been seen once entirely.

At every point in time, Algorithm 3 maintains only $2n$ pointers to input and output streams and a constant number of variables in the memory.

8 Analysis

The output of the string-based filtering step contains the sliding windows of candidate trackers strings. In order to manually validate each string found, i.e., to mark whether it is a tracker or not, it is important to be understand the context in which it appeared, i.e., where in the TCP payload was that string sent on the network. In order to do that, *context building* is performed using the byte offset metadata stored with each string in the output. The output of *context*

Algorithm 2: Removing non persistent strings per user

```
Input: stream u : {u_1, u_2,... } where u_i = is [str, day] tuple. Stream u is sorted on str
Output: stream o : {o_1, o_2,... } of filtered objects, where o_i = [str, day]
1  curr_str ← u_1[str]
2  curr_day ← u_1[day]
3  curr_line, start_line ← 1
4  curr_interesting ← False
5  while not at end of stream u do
6  |   READ next object into u_i
7  |   INCREMENT curr_line by 1
8  |   if u_i[str] ≠ curr_str then
9  |   |   if curr_interesting is True then
10 |   |   |   PrintToOutputStream(start_line, curr_line)
11 |   |   end
12 |   |   curr_str ← u_i[str]
13 |   |   curr_day ← u_i[day]
14 |   |   start_line ← curr_line
15 |   |   curr_interesting ← False
16 |   else
17 |   |   if curr_interesting is True then
18 |   |   |   go to next iteration
19 |   |   end
20 |   |   if u_i[day] ≠ curr_day then
21 |   |   |   SET curr_interesting to True
22 |   |   end
23 |   end
24 end
25 if curr_interesting is True then
26 |   PrintToOutputStream(start_line, curr_line)
27 end
```

building not only rebuilds the tracking string from its sliding windows, but also provides the context line and thereby the header (if it exists) under which the string appears.

To give an idea of the overhead of analysis in the last step, we provide statistics about the size of the filtered content output by the tool that needs to be analyzed. Of the 790 users analyzed, only 244 users had non zero tracker strings at the output with a total size of the context files to be inspected about 815 MB.

8.1 Sanity Check

The motivation behind building the tool was to see whether a generic approach can identify trackers in the network and potentially find new unknown tracking strings. Before moving onto finding novel tracker strings sent over the network, as a sanity check, the tool must at least find the trackers well-known to the security community, i.e., cookies. To see this, we randomly sampled "Cookie" attribute values from the raw traces and looked them up in the final analysis data for the user. We found that for all instances, if the connection in which the cookie occurred

Algorithm 3: Removing common strings

Input: streams $\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_n}$ where each stream $\mathbf{u_i}$ is a list of objects $\{u_{i,1}, u_{i,2}, ...\}$ of form $u_{i,j}$ = [str] for each user i. Each stream u_i is sorted on str

Output: streams $\mathbf{o_1}, \mathbf{o_2}, ..., \mathbf{o_n}$. Each $\mathbf{o_i} = \{o_{i,1}, o_{i,2},... \}$ is a stream of candidate trackers for user i

```
1  INITIALIZE curr_strs = [u_{1,1}, u_{2,1}, .., u_{n,1}]
2  while not at end of all streams do
3      min_string ← SmallestString(curr_strs)
4      INITIALIZE min_streams to empty list []
5      for i going from 1 to n do
6          if curr_strs[i] = min_string then
7              APPEND i to min_streams
8          end
9      end
10     if |min_streams| = 1 then
11         PrintToOutputStream(o_i), where min_streams = {i}
12     end
13     for i ∈ min_streams do
14         READ next object u_{i,j} from stream u_i
15         SET curr_strs[i] to u_{i,j}
16     end
17 end
```

was retained after the connection based filtering step, then the cookie would appear in our final analysis data. Thus our tool at least captures the trackers it is expected to.

8.2 Fine-grained Filtering

In order to further reduce the overhead of manual analysis, we added two more filters; the *URL path filter* removes tracker strings that appeared only in a URL path, and not URL parameters, and the *cookie filter* removes all the tracker strings that appeared in cookies. The motivation behind *URL path filter* is (*i*) web pages with unique names raise the false positive rate, (*ii*) it is not likely that tracking is performed using a specific path in the site's directory structure. We believe that scenarios where users visit a webpage that remains open across multiple days on the user's browser would fall under this category. As an example, the url `http://www.modcloth.com/shop/shoes-heels/tropical-tasting-heel` was captured by our tool with *tropical-tasting-heel* marked as unique. The motivation for *cookie filter* is to find novel tracker strings. Since the security community is aware of cookies being used as tracker strings, we remove them from our analysis. This further reduces the size of files to be analyzed by a factor of approximately 1.7; the total file size now is approximately 483 MB. The total number of users to be analyzed at this stage is 244.

8.3 Results

The analysis of the filtered context files is expensive in terms of analyst time taken. It took about 7-8 hours of manual analysis to go over about 25 out of 244 users. Each line in the context file corresponds to a line from the original content file with the tracker string highlighted. We

Table 4: Potential trackers found

Host	Tracker Name	Context	Category
ui.skype.com	uhash	URL parameter	account specific
dropbox.com	user_id	URL parameter	account specific
symantec.com	useragent	user agent	browser specific
courier.push.apple.com	AppleiPhoneDevice	Non-HTTP: TCP payload	device specific
microsoft.com	USR	Non-HTTP: MSN ping message	account specific
freenode.net	USER, NICK	Non-HTTP: IRC Channel	account specific
jupiter.apads.com	deviceid	HTTP POST payload	device specific
safebrowsing.clients.google.com	wrkey	URL parameter	unknown
s.amazon-adsystem.com	ad-sid, a2, id		
aax-us-east.amazon-adsystem.com	ad-sid, a2		
tags.bluekai.com	a_id, id		
addthis.com	uid		
l.collective-media.net	id		
idsync.rlcdn.com	id		
p.acxiom-online.com	id, uid		
e.nexac.com	id		
d.agkn.com	partner_id		
cspix.media6degrees.com	tpu		
v.admaster.com.cn	c		
aktrack.pubmatic.com	piggybackCookie		
image2.pubmatic.com	piggybackCookie		
d.xp1.ru4.com	u		
livepassdl.conviva.com	uuid		
pixel.quantserve.com	fpa		
api.embed.ly	sid, key, uid		
p.rfihub.com	userid		
loadus.exelator.com	buid2, buid		
static.crowdscience.com	cp0, cp1		
pagead2.googlesyndication.com	uid		
ums.adtechus.com	userid, providerid		
rtb-csync.smartadserver.com	partneruserid		
adnxs.com	xuid		
bizographics.com	bizoid		

print the entire line instead of just the tracker strings in order to obtain the context of each string, i.e., whether it appeared in URL or User Agent or under some other header. We use this context and resources available online (if any) about the string to understand whether the tracker found is a false positive and if not, how is the string being used for tracking, i.e., whether it is account/device/session/browser specific. Table 4 lists all the trackers we found, the domain they were seen from, the context in which they appeared and their category.

Since the cookies have been filtered out for faster analysis, the Table does not include those. Apart from cookies, a large number of other candidate strings were captured by the tool. Interestingly the tool captured a user identifier sent across the network by Skype as a parameter in the URL. The URL after logging into Skype application is of the format `http://ui.skype.com/ui/2/2.1.0.81/en/getlatestversion?ver=2.`

1.0.81&uhash=<uhash>. The parameter uhash [2] is the hash of the user ID, the password and own value of the hash function and is constant if the Skype user is fixed [7]. *uhash* can very well act as a tracker for a user. If the same uhash was seen sent to Skype on two different networks, one can infer that it was the same user on both the networks.

Another interesting discovery was of the Dropbox userid being sent over the network in the clear. The url of the Dropbox application syncing with its server looks like /subscribe?host_int=327016083&ns_map=334914819_1963134969091, 161392333_446837991117,...,230181947_99014429755&user_id= uid&nid=74&ts=1389299857 with a clear text parameter called *user_id* being sent along with its value. Since the Dropbox application regularly syncs with its server, tracking information is being sent out on the network at regular intervals without requiring any user activity.

We also found a few other candidate tracking strings which we believe are interesting but do not have conclusively infer whether they can be used as trackers. For example, our tool captured a parameter called *wrkey*, which appears as a response to safebrowsing.clients.google.com in the URL (example, /safebrowsing/downloads?client=navclient-auto-ffox&appver= 24.3.0&pver=2.2&wrkey=<wrkey>) by Google's safebrowsing server. The documentation provided by Google [6] says that *wrkey* key is sent to the client along with other identifiable information as a private key for confidential communication with the server. We found parameter called deviceId with details of the phone[3] make and service provider, being sent as a part of POST request payload to jupiter.apads.com. Symantec was found to use a random looking user agent for its communication, for example, User-Agent:LmIpWZ1/EyabGJVmgz1sozkFjAUcO/OUgAAAAA. We did not find any proof of it being unique to a machine or a user; however, since it was captured by the tool, the User agent was definitely unique and persistent as seen over the whole range of 15 days. Adaptv also sends out cookie information in the url under the parameter *adaptv_unique_use_cookie*. The tool marked a few User Agents which were unique to the respective users. Interestingly, we also found non-HTTP tracking strings like IRC and MSN messenger ping messages, for example, our tool found "USR 3 SSO I uname@hotmail.com" being sent to microsoft.com with the email address of the user[4] marked as the tracker. Similarly, USER username*irc.freenode.net:purple and NICK nick were sent to freenode.net, where username was the username of the corresponding user and NICK was the nickname used by the user for communication. We also found a device token being sent as a component of Apple's Push Notification (APN) Service. APNs use this token to locate the device and to authenticate the routing of a notification [1]. Other examples of suspicious looking parameters sent on the network are included in the Table 4.

9 Conclusion

Web sites today use tracking techniques to collect information about users' browsing activity. Conventional trackers used for this purpose are well known to the security community and have recently spurred serious concerns about users' privacy. In this work, we build a tool that given the network traces, automatically identifies more such user-identifiable information sent

[2] We used a place holder for the value instead of the actual value since we believe that is could be a user-identifying string

[3] the device was a phone in this case which was clear from other values of the parameters in POST request

[4] We confirmed that this was the real email address of a user in our network

over the network. Scaling the tool to handle large network traces and validating the candidate trackers output by the tool were major key challenges faced by us in this work. We present two key techniques used in our tool to tackle these challenges, namely multi-stage filtering and streaming algorithms. Our tool is successfully able to capture novel trackers sent on the network along with the ones that are well known in the security community. We believe that this tool can provide useful insights into leakage of privacy sensitive information on web today.

References

[1] Apple Push Notification Service. https://developer.apple.com/library/mac/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html.

[2] Bro. https://www.bro.org/.

[3] Nmap free security scanner. http://nmap.org.

[4] Project details for p0f. http://lcamtuf.coredump.cx/p0f.shtml.

[5] Project details for xprobe. http://sourceforge.net/projects/xprobe/.

[6] Safe Browsing API. https://developers.google.com/safe-browsing/developers_guide_v2.

[7] C. M. Arranz. IP Telephony: Peer-to-peer versus SIP. http://kth.diva-portal.org/smash/get/diva2:513157/FULLTEXT01.pdf, 2005.

[8] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[9] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.

[10] C. Kreibich and J. Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, Jan. 2004.

[11] B. Krishnamurthy and C. E. Wills. On the leakage of personally identifiable information via online social networks. In *Proceedings of the 2Nd ACM Workshop on Online Social Networks*, WOSN '09, pages 7–12, New York, NY, USA, 2009. ACM.

[12] B. Krishnamurthy and C. E. Wills. Privacy diffusion on the web: a longitudinal perspective. In *WWW*, pages 541–550, 2009.

[13] J. R. Mayer. Any person... a pamphleteer: Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, 2009.

[14] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association.

[15] Y. Xie, F. Yu, and M. Abadi. De-anonymizing the internet using unreliable ids. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 75–86, New York, NY, USA, 2009. ACM.

[16] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *NDSS*. The Internet Society, 2012.