Sakshi Jain*, Mobin Javed, and Vern Paxson

# Towards Mining Latent Client Identifiers from Network Traffic

**Abstract:** Websites extensively track users via identifiers that uniquely map to client machines or user accounts. Although such tracking has desirable properties like enabling personalization and website analytics, it also raises serious concerns about online user privacy, and can potentially enable illicit surveillance by adversaries who broadly monitor network traffic.

In this work we seek to understand the possibilities of latent identifiers appearing in user traffic in forms beyond those already well-known and studied, such as browser and Flash cookies. We develop a methodology for processing large network traces to semi-automatically discover identifiers sent by clients that distinguish users/devices/browsers, such as usernames, cookies, custom user agents, and IMEI numbers. We address the challenges of scaling such discovery up to enterprise-sized data by devising multistage filtering and streaming algorithms. The resulting methodology reflects trade-offs between reducing the ultimate analysis burden and the risk of missing potential identifier strings. We analyze 15 days of data from a site with several hundred users and capture dozens of latent identifiers, primarily in HTTP request components, but also in non-HTTP protocols.

**Keywords:** privacy, client identifiers, mining network traffic, tracking

DOI 10.1515/popets-2016-0007
Received 2015-08-31; revised 2015-11-19; accepted 2015-12-02.

## 1 Introduction

Websites extensively track users in order to provide personalization and to gather analytics on website usage. In addition, third-party services aggregate information across websites, selling off the resulting user profiles to advertising companies, purportedly to provide users with better targeted ads. This massive tracking raises serious concerns regarding online user privacy because users lack transparency into how their information is gathered, used, and abused.

While the privacy community readily recognizes the wide-spread prevalence of traditional first and third-party *web tracking*, and has over the time developed safeguards in response, websites in turn have adopted more surreptitious mechanisms. For example, *canvas fingerprinting* uses rendering differences to fingerprint browsers, and *ever-cookies* track users even when traditional cookies have been deleted [25]. It remains an open question as to what *other* kinds of tracking information user devices transmit to various servers around the world, unbeknownst to the users.

Tracking information presents a privacy threat not only from the entities with whom users overtly share this information, but also from adversaries who possesses the capability to broadly tap network traffic from various vantage points. For example, recent leaks have revealed that the NSA indeed draws upon tracking information sent over networks for conducting large-scale surveillance [1, 5]. In the context of such adversaries, any persistent, uniquely identifying transmission by a user becomes relevant, whether or not it is actually designed to serve as a tracking identifier.

In this work we consider the problem of discovering hitherto unrecognized ("latent") identifiers—either actual, or potential—by looking for them directly in network traffic. We develop algorithms to discover unique pieces of information sent by network devices. Our work enables the identification of previously unidentified tracking mechanisms: new identifiers either currently in use, or potentially exploitable by those seeking to better track users.

Our main contribution is a methodology to aid in the discovery of latent client identifiers.[1] We base our approach on identifying repeated occurrences of strings observed emanating from only a single network device. To do so efficiently enough to facilitate mining large volumes of network traffic (TBs), we draw upon two key ideas: (*i*) multistage filtering and (*ii*) extensive use of streaming algorithms. Our methodology is *semi-automated* since an analyst must then examine the potential identifiers, along with the context in which they appear, to make a final determination regarding their nature and properties.

***Corresponding Author: Sakshi Jain:** [UC Berkeley, LinkedIn], E-mail: sjain2@linkedin.com
**Mobin Javed:** UC Berkeley, E-mail: mobin@cs.berkeley.edu
**Vern Paxson:** [UC Berkeley, ICSI], E-mail: vern@cs.berkeley.edu

**1** Our code is available here: https://github.com/sakshi-jain/mining-identifiers.

A novel feature of our approach is that it identifies various kinds of identifiers independently of the associated tracking mechanism used (if any), as opposed to prior studies that focus their analysis on particular types of tracking [9, 10, 16, 17, 22, 23, 23, 26, 27]. We demonstrate the effectiveness of our methodology by employing it to find numerous identifiers sent in HTTP headers, URL parameters, and payloads, as well as in non-HTTP messages, such as usernames in MSN and IRC messages. We also find a number of device-specific identifiers sent for mobile devices, both by the OS and by advertising APIs used by various apps.

The ability to extensively mine for identifiers enables further research into novel tracking mechanisms by revealing a set of domains and the corresponding network requests to which clients send identifiers. One can then analyze these sites using host-based instrumentation approaches [9, 10, 15, 19] to determine the tracking mechanisms they use. Our contribution also enables future work on studying the magnitude of privacy threats posed by entities who can tap network traffic and chain the various kinds of identifiers they observe to fingerprint users and build profiles of their activity.

We organize the paper as follows. We begin with related work in § 2. We define what we mean by “identifiers” in § 3. § 4 details the characteristics of the dataset we use in developing and evaluating our methodology. § 5 outlines a naive approach for detecting identifier strings and examines the associated challenges. In § 6, we provide an overview of the various components of our methodology, and in § 7 we present our multistage filtering pipeline along with its implementation. We present the results of our analysis in § 8 and summarize in § 9.

## 2 Related Work

The literature relevant to our work lies in three domains: (1) the design of tracking mechanisms, (2) detecting identifiers and information leakage, and (3) extracting strings of interest from network traffic.

### Design of tracking mechanisms

The first category of work focuses on design of various browser or device tracking techniques. Some studies on device fingerprinting leverage packet-level information to capture subtle differences in host software systems [4, 6, 8] or hardware devices [20]. Eckersley showed that on average, a browser’s version and configuration information, such as screen resolution, plugins and system fonts, contains at least 18.8 bits of uniquely identifying information [13]. Other works study the installation order of browser plug-ins [24] or application level IDs [29] for tracking web clients. In [30], authors compare the effectiveness of host tracking using a variety of identifiers like user agent, IP prefix, cookie ID. Their study shows that servers can still track 88% of returning users even if the users clear cookies or use private browsing. In a study, Krishnamurthy et al. show that third-parties can link Online Social Networks (OSN) identity to tracking cookies, from users’ activities both within and outside the OSN sites [22]. Acar et al. specifically study three persistent tracking mechanisms: canvas fingerprinting, “evercookies”, and cookie-syncing [9]. They show that 5.5% of the top 100K Alexa sites use fingerprinting scripts.

### Detecting Identifiers and Information Leakage

A large body of work studies the leakage of private information and the prevalence of persistent identifiers on the Web, such as fingerprinting and evercookies. Most of these efforts use a combination of techniques. In this section, we broadly classify such efforts based on their methodology into (*i*) HTTP traffic analysis, (*ii*) instrumentated execution, and (*iii*) application code analysis.

**HTTP traffic analysis**: Works in this category study various tracking mechanisms or privacy leaks by analyzing HTTP logs of emulated user traffic. Krishnamurthy et al. showed 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 2005 to 70% in Sep 2008 [23]. They based their study on emulating user browsing activity and examining the leakage of unique identifiers (e.g., OSN usernames) in `Referer`, request URI, or `Cookie` fields in HTTP requests sent to third-party websites. Eubank et al. examined the mobile tracking landscape. They crawled the top 500 Alexa websites using a mobile measurement platform, capturing first- and third-party cookies. Their data showed that mobile and desktop ecosystems share substantially similar top third-party domains [17] with very limited mobile-specific ad networks. Englehardt et al. study the magnitude of privacy threats posed by adversaries who can passively eavesdrop on network traffic [16]. They cluster HTTP traffic by linking unique substrings of third party cookies and show that such an adversary can reconstruct about 62–73% of a user’s browsing history. Other studies highlight the privacy implications of cookie-matching protocols in use by ad exchanges [23, 27].

**Instrumented execution**: These approaches to analyzing tracking involve instrumenting browsers or OS’s to capture leaked tracking information.

*Browser instrumentation:* In [10], the authors develop FPDetective, a framework for the detection and analysis of web-based fingerprints. They base their approach on detecting font probing, and use a combination of Javascript event instrumentation and source code analysis of Flash objects (extracted from network traffic) towards this end. To identify fingerprinters from this data, they use heuristics such as querying at least $n$ fonts. Acar et al. study the prevalence of canvas fingerprinting, evercookies, and cookie-syncing [9]. To detect canvas fingerprinting scripts, they instrument function calls that browsers use to render images, query pixel data, and send this data to the server. They also automate detection of “respawning” to study evercookies. They first use a set of heuristics to extract “identifying elements" from various storage vectors and then check if sites respawn the identifiers on a revisit from a clean-state browser seeded only with Flash cookies of the previous crawl. Their heuristics do not comprehensively catch identifiers; rather, they seek to use conservative rules to achieve low false positive rates.

*Mobile OS instrumentation:* Prior work has examined the leakage of sensitive information and phone-specific identifiers from mobile phones using host-based instrumentation. TaintDroid tracks the leakage of sensitive information by third-party apps [15]. They base their analysis on a predefined list of “sensitive information” consisting of sensor data (location, camera, microphone), phone data (messages, contacts), and device identifiers (IMEI and IMSI). They found that two-thirds of apps in their study send sensitive data suspiciously, such as transmitting device identifiers and geo-location to advertising servers. Han et al. undertook a real-world study of the tracking of 20 participants for three weeks as they used instrumented Android devices [19]. The study found persistent identifiers sent to advertising and analytics servers.

**Application code analysis:** Other work focuses on analyzing key code elements in order to understand the nature of tracking data as well as tracking mechanisms.

Egele et al. studied privacy leaks in iOS using static analysis of 1,400 apps [14]. Like the other work in this domain, they draw upon predefined list of sensitive information, which they extracted by studying the app *Spyphone*. Their work reiterates the findings of previous studies in this domain that devices send IDs to various advertising and analytics servers, and in addition finds examples of surreptitious transmission of address book, browser history, and photo gallery data. Achara et al. study the RATP app for Paris subway using a combination of static and dynamic analysis techniques [11]. They find that in addition to device identifiers, the RATP app transmits a list of apps running on the smartphone to third parties targeting mobile audiences. In [26], the authors analyze the code of three major browser-fingerprinting code providers: Bluecava, Iovation, and Threatmetrix, and identify a number of surreptitious fingerprinting mechanisms in use by the companies.

A common feature across all the identifier-detecting techniques discussed above is that they look for a predefined set of interesting information, such as, cookies, PII, and function calls. Given the reliance on predefined notions of sensitive information, these detection techniques can miss latent identifiers sent via unconventional channels or structured in a hitherto unrecognized fashion. Our methodology can potentially identify such otherwise overlooked identifiers given we focus on pinpointing client-unique data (strings) irrespective of the application protocol or tracking mechanism that employs them. One such example that we find (and discuss in Section 8) is that of device identifiers sent in connections to Apple’s Push Notification (APN) service. We believe that none of the existing techniques have the capability to detect identifiers that are sent by an OS itself since most mobile OS instrumentation work focuses on detecting PII or sensitive information sent by apps only.

### Pattern Recognition on Network Traffic

Other literature mines interesting contents out of network traffic using pattern recognition. Singh et al. built *EarlyBird*, a prototype for automated extraction of worm signatures from network traffic based on highly prevalent strings repeatedly sent between a multitude of hosts [28]. Their work develops counting algorithms to sift through network contents for such strings. *Honeycomb* employs pattern matching to discover new NIDs signatures by looking for the longest common subsequence of strings found in messages involving honeypots [21]. We tackle a somewhat different problem: identifying unique strings sent over a network by a given client.

## 3 Defining Identifiers

The term *tracking* describes the practice by which sites collect information about a user’s activity across one or more sites. In this context, by *identifier* we refer to a piece of unique information that recurs in repeated visits from a given client to the server. Many identifiers do not determine the human user, but a client machine or browser instance.

In our work, we consider the manifestation of identifiers as seen from a *network* vantage point. Consequently, we treat *potential*[2] identifiers as repeating strings observed in traffic sent by only one machine/device in the network. This network-view approach can fail to capture some true identifiers because: (i) the string does not repeat, i.e., only appears a single time for a user during the observation period, (ii) the identifier is user-specific rather than device-specific, and hence repeats

**2** Note, we use *potential* because this definition does not exclusively characterize identifiers.

across multiple devices (belonging to the same user), or (iii) we lack sufficient visibility due to use of encryption.

Further, in this work we seek to find identifiers that *persist*. The longer an identifier continues to uniquely correspond to a machine, the greater its power to track users. In general, there is no *a priori* correct amount of time to require candidate identifiers to span. The value used will trade off opportunities to observe multiple instances of the identifier (which may occur only far apart in time) versus the amount of network traffic available to mine using our methodology. For our present purposes, we chose to only consider identifiers seen over multiple days.

## 4 Data

For our analysis, we use fifteen days of raw network traces captured at the border router of an enterprise network. The network contains 512 unique IP addresses, with four IP addresses corresponding to NATs.

We use the DHCP and NAT logs to resolve individual devices behind the four NATs. DHCP logs provide a mapping between MAC address and private IP address while NAT logs provide a mapping between private and public connection tuples. Using these two logs, for a given timestamp we can map public source IP address and source port to a MAC address behind the NAT. After mapping, we find 290 unique MAC addresses behind the four NATs. In total, there are 790 (500 non-NAT + 290 behind NAT) distinct devices in our dataset.

For simplicity, we consider each non-NAT IP address and each unique MAC address behind the NATs as a separate user for our analysis, even though the same individual user can own a desktop with a fixed public IP address and a laptop connected to a NAT, thereby occupying two users instead of one in our user list.

Our dataset totals 3.5 TB, with an average volume of 4.4 GB per user and 274 MB per user per day.

| | |
|---|---|
| **Size of network traces** | **3.5 TB** |
| **NAT records** | **4.53M** |
| **DHCP records** | **15.8K** |
| **Total days** | **15** |
| **Network** | |
| **Unique IP addresses** | **512** |
| **NAT addresses** | **4** |
| **Unique MAC addresses behind NAT** | **290** |
| **Addresses outside NAT** | **500** |
| **Total users** | **790** |
| **Users with non-zero contents** | **786** |

**Table 1.** Summary of dataset

**Ethical Considerations:** The data comes from a site that collects network traffic, consent for which is included in their user agreement. Since the raw network traces contain potentially sensitive information, we provide our research code to site personnel who run it locally on the traces. The code distills data down to a set of candidate identifier strings that we analyze. The IRB we work with classified our study as not human subjects research, as it is does not involve interacting or intervening with individuals; rather, we measure the behavior of devices.

## 5 Key Challenges

In principle, detecting strings that are unique and persistent in network traffic is quite straightforward. In this section, we discuss a naive algorithm and highlight the key challenges associated with it, thereby motivating the need to develop a more sophisticated approach. We then discuss the general approaches that we use to address the challenges.

```
Algorithm 1: Naive approach for finding unique strings
   Input: $user_1, user_2, ...user_n$, where $user_i$ is a list of
          strings for user $i$
   Output: a 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 $user_i \cap$ strings in $string\_count$
12 end
```

### 5.1 Naive Approach

Algorithm 1 shows the pseudo code for obtaining the set of unique strings for each user in a network. The algorithm works as follows: for each user $i$, distill their contents from the network traffic 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 for which each substring occurred. 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. This leaves us with a global list of just those substrings for which we found exactly one user. In order to associate these substrings to the desired user, we output the intersection of a 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 that persist across a certain time window (say across days), 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 time periods during which they occurred. As mentioned above, for our present work we restrict ourselves to identifiers that manfiest across multiple days. From here on, for convenience we discuss identifying the "days" associated with a given string, rather than the more general notion of "time periods".

Once we apply the persistence check, this approach leaves us with the candidate identifier strings for each user.

## 5.2 Challenges

The approach described above clearly achieves our objective of weeding out the strings that are not unique or persistent across a certain time window. However, in its current form, Algorithm 1 is far from practical, both in terms of time and space requirements. Here, we go over the key challenges posed by this approach.

**Scaling.** The naive algorithm extracts substrings of all possible lengths from each user's network traffic. Although this is ideal for finding all identifier strings, the memory requirement for storing all possible substrings is about O($\sum_i L_i^2$), where $L_i$ is the total number of bytes in the network trace of user $i$. This clearly does not scale well for the problem at hand, where we are dealing with terabytes of data.

**Validation.** Since the naive algorithm is based only on the properties of uniqueness and persistence, it can result in a large number of *false positives*: strings that are not identifiers but marked as candidate identifiers by the algorithm. It can exhibit false positives due to:

*(i) Server connections unique to a user:* If a user regularly establishes connections with a server $S$ to which no other user in the network connects, then with high probability any content sent to and from this server would be unique to the user, and the naive algorithm would wrongly mark it as a potential identifier.

*(ii) Server content unique to a user:* Consider a scenario in which a user connects to a server $T$, to which other users in the network also connect, but the user repeatedly visits a unique webpage. (For example, a song on a popular online radio station). Any content from this web page will appear unique to the user, leading us to wrongly mark it as a potential identifier.

*(iii) Encrypted traffic:* For correctly implemented encryption, any strings seen repeatedly in encrypted traffic will arise only due to chance.

For our approach, validating the output of the algorithm is a manual process; an analyst must make a decision whether each output string truly represents an identifier. Consequently, a high false positive rate will lead to a heavy burden on the analyst.

## 5.3 More efficient approaches

To tackle the above challenges, we employ three techniques. First, we introduce the use of *sliding windows* to reduce the storage requirements for tracking strings, at the cost of potentially missing some latent identifiers. Second, we structure the search for unique strings using *multistage filtering*, which aims to winnow down the required processing using simple rules that in general should not forgo many opportunities to discover identifiers. Finally, we implement most of the stages in our analysis pipeline via *streaming algorithms*, which help reduce the required memory footprint even when running on large volumes of strings.

**Sliding Windows.** We note that instead of extracting substrings of all possible lengths, we can process the data by using sliding windows of a fixed length *k*. By choosing a sliding window of fixed length, we reduce the total size of substrings from quadratic to linear in the size of the input trace. We can then later reconstruct full identifier strings of length greater than *k* from the respective sliding windows by storing associated metadata. However, we will necessarily lose all identifier strings of length less than *k*, and can also potentially lose some identifier strings of length greater than *k*, since the substrings of length $k$ of a longer identifier are not necessarily unique.

**Multistage Filtering.** The idea behind multistage filtering is to progressively remove any content that has a low probability of enabling us to discover latent identifiers, and then to apply the constraints of uniqueness and persistence to the smaller set of remaining data. At each stage, we consider trade-offs between computation gain coupled with the reduced false positive rates obtained by using the filter on one hand, versus lost opportunities to discover true identifiers on the other. This approach of progressively winnowing down the input stream not only makes our approach more scalable but also reduces the overhead of manual analysis for validation.

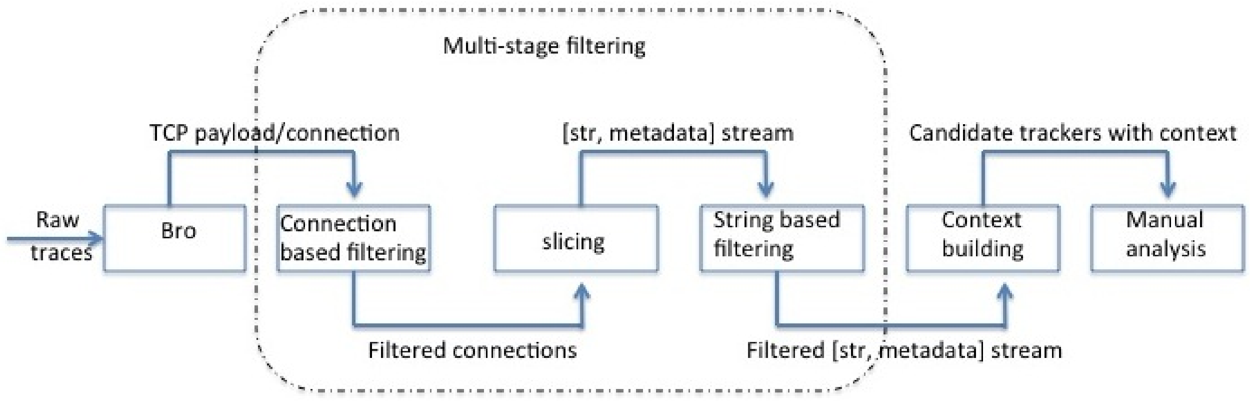

**Fig. 1.** Architecture diagram for our data processing pipeline.

**Streaming Algorithms.** We develop streaming algorithms that can efficiently perform filtering with more modest memory requirements than naive counting algorithms. Streaming algorithms typically make only one pass over the data, using memory much smaller than the total input size, and thus prove particularly advantageous when the entire dataset cannot fit in memory.

That said, we note that one stage of our processing pipeline requires sorting, which cannot be done in a streaming fashion. We discuss steps we take to streamline the sorting stage in the next section.

## 6 Architecture Overview

We base our approach on a series of processing and filtering stages. Figure 1 shows the entire pipeline schematically. In this section, we give an overview of the different stages, and discuss the kind of data and metadata retained as the data moves through the pipeline.

**Preprocessing:** We use the network traffic analysis tool `Bro` [3] to extract TCP payloads from the network traces. This step produces content files along with metadata on the corresponding connection for each user. Each content file corresponds to the stream of TCP payload data for that connection.

**Connection-based Filtering:** We use the metadata associated with each connection, i.e., information contained in the four-tuple ⟨sourceIP, sourcePort, destinationIP, destinationPort⟩, to either filter it out or feed it to the next step. We discuss the details of connection-based filters in § 7.1.

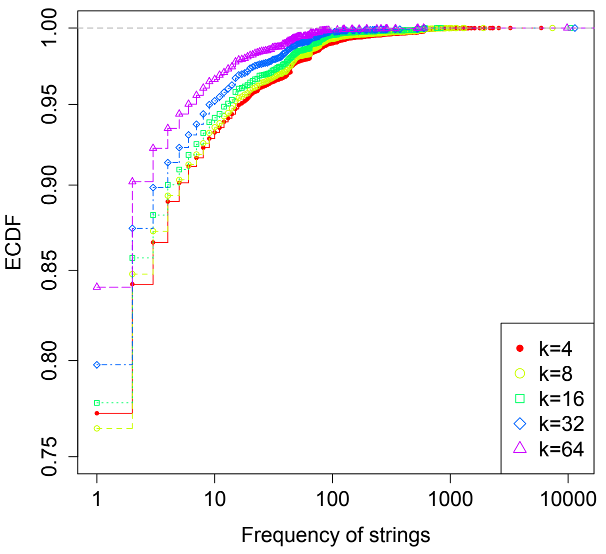

**Fig. 2.** Log-linear plot of distribution of count of unique strings vs. their frequency of occurrence in one user's trace

**Slicing:** Of the connections that remain, we slice the data into sliding windows of $k$ bytes, which we refer to as "strings" here onwards. For filtering in subsequent stages and to rebuild the original context from the sliding windows for eventual analysis, we associate some metadata with each instance of a string. 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. This stage generates a stream of $k$-byte strings and the associated metadata for each user.

*Choosing k: k* sets the size of the string fragments that we use for processing data. We use repeated instances of such fragments for filtering. A smaller $k$ is better from the perspective of memory requirements, but results in larger false negatives. For example, consider $k = 4$, an interesting identifier which takes values "Michael" and "Michelangelo" for two different users would be filtered out due to repetition of "Mich". A large $k$ value, on the other hand can result in both false negatives (missing out identifiers of length $< k$) and higher false positives. For example, if $k = 20$, then "ref=google.com&uid2=" and "ref=google.com&uid3=" sent by two machines would be falsely marked as two unique identifiers by our flow. In order to capture the trade-off between possibly interesting identifiers lost and memory consumption, we computed sliding windows of various lengths on a sample user trace. Figure 2 shows a CDF of the count of distinct strings (hence potential identifier strings) vs. their frequency for the values of $k$ in the set {4, 8, 16, 32, 64}. From the figure, the number of distinct strings, hence the sample space for all processing, does not change much going from $k = 64$ to $k = 8$ while giving a large reduction in total data size. Hence, we choose $k = 8$ as the size of sliding window that can serve well for tracking identifer strings.

**Bucketing and Sorting:** Before processing these streams further, we sort the list of strings for each user. We reduce the memory requirements and $O(n \log n)$ running time for doing so by partitioning the strings into 256 different buckets based on the ASCII encoding of their first character. For example, we place all strings starting with "A" in bucket number 65. (The actual structure for doing so is a directory in the file system with 256 subdirectories.) Since in string-based filtering, whether we filter out a string or forward it to the analysis stage solely depends on the various statistics corresponding to the particular string, bucketing does not interfere with the processing. 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 pipeline. The memory footprint of our algorithms mainly depends on how efficiently we can sort each of these files. Note that since we perform sorting for each user and each bucket separately, the processing remains tractable as the number of users increases, the bottleneck being the user with the most content.

**String-based Filtering:** The next step, *string-based filtering*, ensures uniqueness and persistence of the remaining strings. To efficiently filter, we use streaming algorithms, developing algorithms for removing both non-persistent strings and strings common across users. Note that although streaming algorithms help us reduce the memory consumed by our methodology, the disk storage space required for subsequent processing steps increases. We discuss the details of our string-based filtering algorithms in § 7.2.

| Filter Name | Median | IQR |
|---|---|---|
| **Encrypted traffic** | **1.0** | **314.5** |
| **Server-originated traffic** | **2.0** | **1.3** |
| **Unique server data** | **1.2** | **160.0** |

**Table 2.** Filtering strengths of connection-based filters.

**Context-building & Analysis:** The output of these steps is a set of candidate identifier strings for each user. We analyze these candidates manually for validation. To assist manual analysis, we use the associated metadata to rebuild the original strings from their sliding windows. This facilitates the validation process by providing the context in which the string appeared in the original trace—for example, whether it occurred in part of a URL or Cookie. We discuss the context building stage and analysis in § 8.

## 7 Multistage Filtering

In this section we describe the connection-based and string-based filtering stages. We discuss what each filter achieves and the trade-offs we considered when selecting it. Figure 3 shows a schematic diagram summarizing the various filters.

We measure each filter's efficacy in terms of data reduction by computing its *filtering strength*, which we define for a user as the ratio of filtered TCP payload versus total size of input TCP payload. Note that our input data contains externally initiated connections to servers inside the network as well as machines which are not actively used by users. The volumes associated with different systems varies widely. To robustly characterize the typical range of values we observe, we describe the overall filtering strength in terms of the median and interquartile range (IQR) of individual user filtering strengths. Note that each filter's strength depends on which filters ran previous to it. For uniformity, we provide the *standalone filtering strength* of each filter. If $N$ is the total size of raw content files, the standalone filtering strength is $N$ divided by the size of the output when we apply the filter under consideration directly to the raw content files, with no other preceding filters.

### 7.1 Connection-based Filtering

This stage performs filtering based on the properties of a connection, identified by the four-tuple ⟨sourceIP, sourcePort, destinationIP, destinationPort⟩. We remove three kinds of content

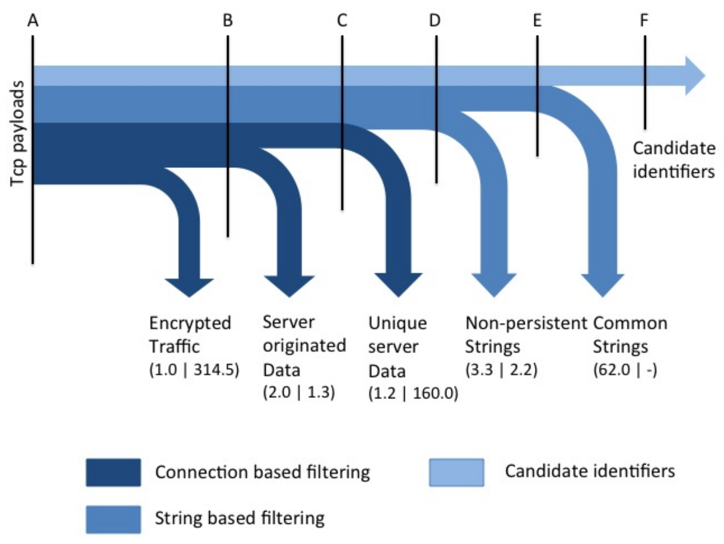
A
B
C
D
E
F
Tcp payloads
Candidate identifiers
Encrypted Traffic (1.0 | 314.5)
Server originated Data (2.0 | 1.3)
Unique server Data (1.2 | 160.0)
Non-persistent Strings (3.3 | 2.2)
Common Strings (62.0 | -)
Connection based filtering
String based filtering
Candidate identifiers

**Fig. 3.** The connection-based and string-based filters as they appear in the processing pipeline. Each filter is annotated with (median | interquartile range) values

from our analysis: (*i*) encrypted connections, (*ii*) data sent by servers to clients, and (*iii*) connections to servers visited by only a single user. Table 2 gives the filtering strength provided by each of these filters.

**(i) Remove encrypted connections.** We identify encrypted connections as those with a server port of 22 (SSH) or 443 (HTTPS). As discussed earlier, with properly implemented encryption, a content-based technique like ours should not be able to locate any recurring identifiers in such payload. Note however that this filter also foregoes discovering potential latent identifiers sent in plain text as a part of the encryption protocol itself, such as in TLS handshakes.

The overall filtering strength of encryption-based filter is given by the interquartile range of $\approx$ 314.5 and median of 1.0, implying that at least half of the users did not have SSH/HTTPS traffic.

**(ii) Remove server-originated bytestream.** We remove all responder bytestreams, i.e., all data sent by servers to clients. While some client identifiers can in fact originate in traffic sent by servers (such as HTTP cookies), removing server-side traffic should not cost us opportunities to find these identifiers since they should also recur in client-side traffic. In addition, client-originated data is usually significantly smaller in volume than server-originated data, providing a significant filtering gain.

The filtering strength of the server-based filter has a median per-user value of 2.0, with an IQR of about 1.34. The value is much lower than we expected since there are multiple connections made to servers within the network 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 4,674.

**(iii) Remove unique server connections.** If only a single user in the network connects to a particular server, then we filter out all of the connections between the user and that server. Unlike the previous two filters, which we can apply independently to each user, this stage depends not only 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. By doing so, it is possible that two users access the same domain using different server IP addresses which would lead us to mark each access as unique, and thereby wrongly filter out the content from the domain for both the users.

This filter provides a major reduction in the false positive rate. Any content unique to the server (e.g., specific URLs or web page content) would almost surely be marked wrongly by our methodology as a candidate identifier for the corresponding user. Similarly to the previous filter, we do potentially lose the opportunity to discover some latent identifiers.

The filtering strength of this filter largely depends on the connections made by the *other* users in the network. For networks with many users, the number of unique servers per user should diminish compared to networks with fewer users. For our data, the overall filtering strength of this filter is given by the interquartile range of about 160 and a median of 1.2.

| Filter Name | Median | IQR |
|---|---|---|
| Non-persistent strings | 3.3 | 2.2 |
| Common strings | 62.0 | - |

**Table 3.** Filtering strengths of string-based filters.

## 7.2 String-Based Filtering

In this stage, we perform filtering based on the properties of individual strings in client traces. Recall that the input to this stage is a list of *k*-byte strings and associated metadata, generated using the *slicing* component (per § 6) on the connections left after the *connection-based filtering* stage. We apply two filters in this stage: (*i*) remove non-persistent strings, and (*ii*) remove strings common across users, corresponding to the two conditions of *persistence* and *uniqueness* as required by our definition of identifiers. Table 3 provides the filtering strength statistics of each of these filters, computed over the set of users who had non-zero bytes at the beginning of this filtering step (345 out of 790).

**(i) Remove non-persistent strings.** This stage weeds out all strings that do not repeat over multiple time windows. The bigger the time window, the stronger is the identifier in terms of its tracking power. In our present work, we use 1 day for our time window. Thus, we only retain strings observed on at least 2 different days. We can apply this filtering stage independently to each user's list of strings, since the persistence condition is independent of the presence of other users in the network.

Note that in this stage we automatically filter out any string that occurs only once in a user's trace. This might mean losing some true identifiers that happen to appear only once in our dataset due to its limited size. We accept this trade-off, however, since without multiple occurrences we cannot validate whether the string is in fact an identifier.

The interquartile range for the filtering strength of this stage is 2.2, with a median of 3.3 and an average of 1.5.

*Algorithm:* Algorithm 2 provides pseudocode for the streaming algorithm for this filter. Input data is read as a stream of [*str, day*] tuples. Since we sort the input according to *str*, information about all the days the string manifested appears adjacent to each other in the stream, making it possible to process the data in a streaming fashion. To decide whether to filter out a given string, the algorithm maintains information about the string value ($curr\_str$), day ($curr\_day$) and its start index ($start\_index$) from the first time the string appeared in the stream. As the stream proceeds, either the same string will reappear on the same or different days, or a new string will 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. Otherwise, the algorithm prints it to the output filtered stream. Note that the output stream remains sorted on string value due to the sorted input stream and the streaming nature of the algorithm.

At any point in time, Algorithm 2 maintains only a constant number of variables and 2 cursors 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.

**Algorithm 2:** Removing non-persistent strings per user

```
Input: stream u : {$u_1, u_2$,... } is for one user 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]
$curr\_str \leftarrow u_1[str]$
$curr\_day \leftarrow u_1[day]$
$curr\_index, start\_index \leftarrow 1$
$curr\_interesting \leftarrow$ False
while not at end of stream u do
    READ next object into $u_i$
    INCREMENT $curr\_index$ by 1
    if $u_i[str] \neq curr\_str$ then
        if $curr\_interesting$ is True then
            PrintToOutputStream($start\_index$,
            $curr\_index - 1$)
        end
        $curr\_str \leftarrow u_i[str]$
        $curr\_day \leftarrow u_i[day]$
        $start\_index \leftarrow curr\_index$
        $curr\_interesting \leftarrow$ False
    else
        if $curr\_interesting$ is True then
            go to next iteration
        end
        if $u_i[day] \neq curr\_day$ then
            SET $curr\_interesting$ to True
        end
    end
end
if $curr\_interesting$ is True then
    PrintToOutputStream($start\_index, curr\_index$)
end
```

**(ii) Remove strings common across users.** In this stage, we filter out any strings that are observed across multiple users. This filter follows from our definition that identifiers must be unique to a user. Note that in this stage, we also filter out strings which are true identifiers 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 using only a single device, our methodology will capture the identifier. Thus, in this stage, we might lose a few user-specific identifiers but we do not lose any device-specific identifiers.

The median filtering strength for this filter is 62.0, with an average of 26.7. The 25th percentile is 16, while the 75th percentile is $\infty$, indicating that this step filtered out all of the bytes of at least 25% of users.

*Algorithm:* Algorithm 3 provides pseudocode for the streaming algorithm for identifying unique strings for each user. The input to this stage is $n$ streams, $\mathbf{u^1}, \mathbf{u^2}, ..., \mathbf{u^n}$, where $n$ corresponds to the number of users, and each stream $\mathbf{u^i}$ is the output of the previous persistence filtering stage for user $i$. Since information about the day associated with each string is not relevant for this stage, we represent an object of a stream simply by the string value $str$. Each input stream $\mathbf{u^i}$ (output stream from persistence filter) to this stage is sorted on the ASCII encoding of $str$. The algorithm maintains a list of cursors $curr\_strs$ to the current $str$ in each stream $\mathbf{u^i}$. We also maintain a separate list of cursors to the output stream of each user, $\mathbf{o^1}, \mathbf{o^2}, ..., \mathbf{o^n}$, to which to print elements that survive the filtering. At every stage, the algorithm generates a list of streams, $min\_streams$, which point to the smallest string (in terms of the ASCII collating sequence) in $curr\_strs$. If this list contains more than one stream, the algorithm filters out the given string and proceeds to the next string. If, on the other hand, $min\_streams$ list contains only a single stream, say $\mathbf{u^i}$, then the smallest string was unique to the stream $\mathbf{u^i}$, and we copy all occurrences of the to $\mathbf{o^i}$ and load the next string in $\mathbf{u^i}$ into $curr\_strs$.

At every point in time, Algorithm 3 maintains in memory only $2n$ cursors, $n$ to input and $n$ to output streams and a constant amount of storage for variables.

**Algorithm 3:** Removing common strings

```
Input: streams $\mathbf{u_1}, \mathbf{u_2}, ..., \mathbf{u_n}$ where each stream $\mathbf{u_i}$ is,
       for each user $i$, a list of objects $\{u_{i,1}, u_{i,2}, ...\}$
       of form $u_{i,j}$ = [str] where str is the value of
       sliding window. 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
       identifiers 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      $min\_string \leftarrow$ 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($\mathbf{o_i}$), where
           $min\_streams = \{i\}$
12     end
13     for $i \in min\_streams$ do
14         READ next object $u_{i,j}$ from stream $\mathbf{u_i}$
15         SET $curr\_strs[i]$ to $u_{i,j}$
16     end
17 end
```

## 8 Analysis

In this section we analyze the results produced by applying our methodology to our 15-day dataset. We begin by sketching the general approach we take to the analysis. We then discuss filtering steps used for the manual assessment stage to reduce the analyst burden, followed by summarizing what the process ultimately discovered.

### 8.1 Facilitating Manual Analysis

Our processing pipeline produces *candidate* identifiers: those that in the context of the analyzed trace appear to potentially function as unique identifiers. However, assessing the actual nature of the candidates requires manual assessment, since the determination requires considering additional context beyond that which our algorithms can incorporate. This includes the semantics of the protocol elements in which the candidate is embedded (for example, whether and where it appeared in a URL). Because of the need for this manual stage, we term our methodology as *semi-automated*, rather than automated.

We can, however, automate some of the development of the surrounding context to facilitate the manual assessment. To create a data format consumable by an analyst, we use the byte-offset metadata that we recorded for each candidate identifier string in order to (1) expand the *k*-byte string to the full size of the identifier, and (2) provide additional bytes before and after the full identifier to aid in determining its usage.

For the first step, we note that if we have an $m$-byte full identifier ($m \geq k$), then our algorithms will most likely flag each of its *k*-byte substrings as candidate identifiers. (It may

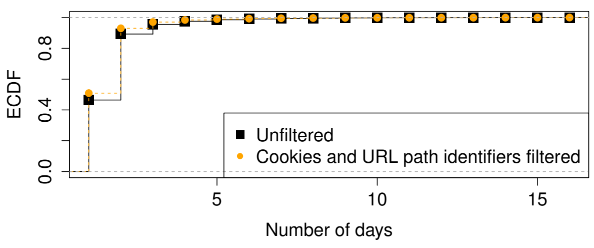

**Fig. 4.** Empirical CDF of persistence of reconstructed identifiers over days, after the context-building stage.

| **Total number of identifiers** | **5,945,993** |
|---|---|
| **HTTP** | |
| `Cookie` **header** | **50.38%** |
| **URL parameters** | **20.41%** |
| **URL path** | **13.16%** |
| `User-Agent` **header** | **0.94%** |
| **Other headers and payloads** | **11.36%** |
| **Non-HTTP** | **3.73%** |

**Table 4.** Major categories of contexts for candidate identifiers

filter out some of them due to happenstance matches between them and strings sent by other clients, per Algorithm 3.) Thus, by merging adjacent or overlapping candidate identifiers, we can produce full *reconstructed identifiers* in a fashion that is robust to the absence of a few *k*-byte substrings, as long as the candidates include some degree of overlap or adjacency.

Given the reconstructed identifiers, we then produce one context file for each user. At this point in the development of our work, we focus on analyzing potential identifiers in ASCII-oriented traffic, since this is much easier for an analyst to deal with. Accordingly, each entry in a context file corresponds to a line (i.e., delineated by newlines) from the original content file, with the reconstructed identifier string(s) highlighted.

For our manual assessment of the candidates, we then use this context (and any online resources we can find relating to the terms appearing in the line) to determine whether the candidate identifier reflects a false positive or a true identifier. If the latter, then we assess how the string is used for tracking, i.e., whether it is account-, device-, or browser-specific.

## 8.2 Analysis Dataset Characterization

Of the 790 users in our dataset, 244 produced at least one identifier string in the final output. The context files at this stage totaled 815 MB (about 0.08% of the original dataset): 1.4M context lines with a total of 5.9M candidate identifier strings.

While this represents a major reduction in volume, the analyst still faces an intractable task in assessing each candidate individually. We thus perform additional filtering based on the characteristics and context of the reconstructed candidate identifiers to further reduce the analysis burden.

First, we note that a very large proportion of our candidate identifiers appear to indeed reflect correct discoveries—as they appear in HTTP `Cookie` headers—but also uninteresting findings given that these do not constitute "latent" identifiers, as they are well-known. While it is encouraging to see that our methodology locates these, they do not merit further investigation on the part of the analyst.

Along these lines, we can categorize the different general contexts in which the candidate identifiers appear, per Table 4. `Cookie` predominates, with HTTP URLs making up another large portion, and `User-Agent` being the second most popular HTTP header. We also find candidate identifiers manifesting in non-HTTP packets.

We also consider the degree of persistence manifested by the reconstructed identifiers, per the distribution shown in Figure 4. 45% of the reconstructed identifiers in fact lose the persistence property of their constituent *k*-byte substrings. While our processing pipeline assured that the substrings necessarily manifested in at least two separate days, the reconstructed identifier itself often does not. This arises because the substrings might not necessarily repeat in an adjacent or overlapping fashion with one another during the separate days. Consider an example from our data: our algorithm reconstructs the 8-character sliding windows of the string "*first-open-streets-event*" to "*first-open-streets-event/*" in the context of a `Referer` header (i.e., this was the value at the end of a `Referer` URL), but to "*first-open-streets-event/; __j*" in the context of a `Cookie` header. Only the longer one happens to repeat over multiple days.

## 8.3 Candidate Identifier Filtering

Given the above considerations, to aid the manual analysis phase we apply three additional filters:

– *Cookie filter:* Remove all instances of candidate identifier strings that appear in HTTP `Cookie` headers. Note that we also remove instances of such identifiers occurring in other contexts, since these are unlikely to prove interesting given we have already identified the usage within the well-known context of cookies.
– *URL Path filter:* Remove identifier strings that appear in the *path* component of URLs in HTTP requests. The rationale behind this filter is that it is relatively unlikely

that servers use specific paths in the site's directory structure to serve as identifiers, but we find that strings occurring in URL paths significantly increase our rate of false candidates, due to users visiting webpages that become reloaded across multiple days. As an example, our methodology marked as a candidate identifier the string "*tropical-tasting-heel*", occurring in the path component of the URL http://www.modcloth.com/shop/shoes-heels/tropical-tasting-heel. The filter removes this sort of candidate.

– *Persistence filter:* Remove identifier strings that appear on only 1 or 2 days. We chose a threshold of at least 3 days based on a preliminary investigation that found that many of the identifiers appearing on only one or two days represent false positives (for example, fragments of URLs in browser reloads), and on the basis that the more interesting and powerful identifiers will manifest more often in 15 days of network traffic.

Applying these three filters sequentially reduces the analyst work in terms of context lines by 33% after the first filter, 63% after the second, and 85% after the application of all three.

## 8.4 Results

Focusing our analysis on the set of identifiers that repeat across at least three days, and do not occur in `Cookie` headers or URL paths reduces our original 5.9M candidate identifiers to a set of 12.3K that we need to validate via manual analysis.

The volume becomes tractable, as follows. We conducted our manual assessment using an iterative process. Upon inspecting a candidate (reconstructed) identifier, along with making our assessment regarding its significance, we devised a rule that would match other instances of the same type of identifier. We applied the rule to tag and remove those from requiring further assessment. All in all, for our dataset we wound up devising 47 such rules, enabling us to manually consider all of the different classes of candidate identifiers in this set.

Clients sent the candidate identifiers to a total of 1,322 different servers, with 661 each in the HTTP and non-HTTP category.[3] Analytics and advertising companies heavily dominate the HTTP domains, and 71% of these domains share a candidate identifier with at least one other domain. We found that in majority of cases, what is shared is not a true identifier, but instead a fragment of a URL embedded in the tracking domains (such as a Google Analytics inclusion). This scenario represents the major remaining source of false positives in our dataset. (Note that the URL path filter did not remove these candidates because the URL fragments appear in the parameters of other URLs, not in their path.)

Table 5 provides a sample of the identifiers we found, the associated domain, the context in which they appeared, and their category. We now discuss each of the different types.

### 8.4.1 HTTP identifiers

**Application-specific:** The first category is identifiers sent by applications other than browsers. For example, Skype sends a user identifier *uhash* in a URL 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* is a hash of the user ID, their password, and a salt, and remains constant for a given Skype user [12]. *uhash* can very well act as an identifier for a user; a monitor who observes the same value from two different clients/networks can infer that it reflects the same user on both.

Another example in this category is a Dropbox *user_id* sent as a URL parameter. We discovered that since the Dropbox application regularly syncs with its server, it sends out this identifier—surprisingly, every minute—without requiring any user action.

**Mobile devices:** Our methodology enabled us to discover that the Apple weather app sends IMEI and IMSI numbers in POST requests to `iphone-wu.apple.com`. We can recognize these as such, because the parameter name in the context clearly names them as IMEI and IMSI; the value also matches the expected format for these identifiers. Other apps also send a number of device identifiers, such as phone make, advertising ID,[4] SHA1 hashes of serial number, MAC address, and UDID (unique device identifier) across various domains, such as s.amazon-adsystem.com, jupiter.apads.com and ads.mp.mydas.mobi. The iOS and Android mobile SDKs provide access to these identifiers.

**Cross-domain:** We also found instances of tracking information sent to multiple domains under the same as well as different parameter names. One identifier was sent to 18 distinct hosts as part of both URLs and cookies: for example, to ib.adnxs.com, sync.adap.tv, sync.mathtag.com, 360yield.com and rlcdn.com as a part of URL with parameter names `code`, `uid`, `mt_exuid`, `external_user_id`

**3** For HTTP, we used the registered domain of the HTTP `Host` header field. For non-HTTP, we used the server IP address to which the identifier was sent.

**4** A randomly generated, resettable unique identifer, per http://goo.gl/P8rPoI and http://goo.gl/HBo99J.

| Domain | Identifier 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 |
| s.amazon-adsystem.com | sha1_mac,adId,sha1_serial,sha1_udid | URL parameter | mobile device-specific |
| ads.mp.mydas.mobi | mmdid,mm_mmdid | URL parameter | mobile device-specific |
| iphone-wu.apple.com | imei | URL parameter | mobile device-specific |
| | imsi | HTTP header: `X-Client-ID` | |
| safebrowsing.clients.google.com | wrkey | URL parameter | third-party domain potentially browser-specific |
| 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 | | |

**Table 5.** A sample of the identifiers captured by our methodology.

and `partner_uid`, respectively. Another identifier sent as a URL parameter to multiple hosts is *google_gid*, which is a Google user ID corresponding to a user's Google cookie [18]. Google sends this identifier to ad bidders who then use the identifier to synchronize their cookies corresponding to this user [27]. While we observe many instances of cross-domain identifiers in our dataset, we find *google_gid* particularly interesting as it provides insight into the cookie-syncing used in the ad ecosystem.

**Other:** We identified a few other candidates for which we were unable to conclusively determine whether they can be used as true identifiers. For example, we identified a parameter called *wrkey* sent to safebrowsing.clients.google.com in a URL of the format `/safebrowsing/downloads?client=navclient-auto-ffox&appver=24.3.0&pver=2.2&wrkey=<`*wrkey*`>`. According to Google's documentation, a browser wishing to receive an integrity MAC protecting responses from the Safe Browsing server first requests a *wrkey* (wrapped key) from the server, and transmits it in subsequent requests [7]. Similarly, Symantec sends a random-looking `User-Agent` in its communication (for example, `User-Agent: LmIpWZ1/EyabGJVmgz1sozkFjAUcO/OUgAAAAA`). We did not find any documentation confirming that this is unique to a client, but our analysis did find multiple unique and persistent Symantec `User-Agents` in our dataset.

| | | |
|---|---|---|
| **Total number of non-HTTP identifiers** | | **1,320** |
| **Total number of distinct ports seen** | | **30** |
| **Ports** | | |
| **2121** | **FTP Proxy** | **32%** |
| **53** | **Domain Name System (DNS)** | **22%** |
| **5223** | **Apple Push Notification Service** | **21%** |
| **5222** | **XMPP client connection** | **7%** |
| **6667** | **Internet Relay Chat (IRC)** | **7%** |

**Table 6.** Top five ports for non-HTTP identifiers.

### 8.4.2 Non-HTTP identifiers

Candidate non-HTTP identifiers constitute 11% of the fully filtered dataset. Table 6 shows the distribution of non-HTTP identifiers over the top five ports. Port 2121 is used for secure file transfers, and port 5222 by instant-messaging applications like iChat, Google Talk, and Jabber. Since the candidate identifiers and the surrounding context for these are not human-readable, without significant additional investigation we were unable to determine whether the information sent to these ports indeed represent true identifiers. The identifiers sent to port 53 result from two servers in the dataset involved in DNS zone transfers, and thus reflect false positives.

**Device identifiers sent by iOS/OSX:** We found instances of device identifiers sent on port 5223. Apple devices use this port to maintain a persistent connection with Apple's Push Notification (APN) service, through which they receive push notifications for installed apps. An app-provider sends to an APN server the push notification along with the *device token* of the recipient device. The APN server in turn forwards the notification to the device, identifiying it via the *device token* [2]. This *device token* is an opaque device identifier, which the APN service gives to the device when it first connects. The device sends this token (in clear text) to the APN server on every connection, and to each app-provider upon app installation. This identifier enabled us to identify 68 clients in our dataset as Apple devices. The devices sent their *device token* to a total of 407 IP addresses in two networks belonging to Apple (17.172.232/24, 17.149/16).

**User-specific identifiers:** We also found non-HTTP identifiers sent in IRC and MSN messenger ping messages. For example, we observed MSN messenger sending the string "`USR 3 SSO I` *uname*`@hotmail.com`" to microsoft.com with the email address of the user as the identifier. Similarly, the IRC messenger sends messages of the format: `USER` *username* `* irc.freenode.net :purple` and `NICK` *nick* to freenode.net, where `username` is the username of the corresponding user and `NICK` is the nickname used by the user for communication.

## 9 Summary

Websites widely track users using *identifiers* to gather analytics on their browsing activity and to provide personalized web content. Such massive tracking raises serious concerns around user privacy due to lack of public knowledge about how this information is gathered and used. In addition, network eavesdroppers can use the presence of such identifiers—even those not specifically designed to facilitate tracking—to surveil user activity. While the security community is well-versed in the workings of conventional tracking mechanisms, it remains an open question regarding to what extent other, hitherto unrecognized mechanisms exist, either potential or actual, that can be used to track users or their devices.

As a step towards facilitating the discovery of such latent identifiers, we develop an application-independent methodology to semi-automatically locate identifier strings by trawling through raw network traffic. Doing so requires overcoming key scaling challenges, which we address using multistage filtering and streaming algorithms. We analyze fifteen days of raw traces from an enterprise network with a few hundred users and find numerous first and third-party identifiers sent in HTTP headers, URL parameters, and payloads, including in non-HTTP messages such as device IDs sent by iPhones in messages to Apple. Our methodology holds promise for helping researchers develop further insight into tracking information being sent to servers around the world, unbeknownst to users.

## Acknowledgments

Our special thanks to David Wagner for discussions on various aspects of this work.

This work was supported by the Intel Science and Technology Center for Secure Computing, the U.S. Army Research Office under MURI grant W911NF-09-1-0553, and by the National Science Foundation under grant CNS-1237265. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

## References

[1] Angry Birds and leaky phone apps targeted by NSA and GCHQ for user data. http://tinyurl.com/nfnd79z. Online. Sep, 2015.

[2] Apple Push Notification Service. http://tinyurl.com/qebsrfy. Online. Sep, 2015.

[3] Bro. https://www.bro.org/. Online. Sep, 2015.

[4] Nmap Free Security Scanner. http://nmap.org. Online. Sep, 2015.

[5] NSA uses Google cookies to pinpoint targets for hacking. http://tinyurl.com/oshq22e. Online. Sep, 2015.

[6] p0f. http://lcamtuf.coredump.cx/p0f.shtml. Online. Sep, 2015.

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

[8] Xprobe. http://sourceforge.net/projects/xprobe/. Online. Sep, 2015.

[9] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2014*.

[10] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. Fpdetective: Dusting the web for fingerprinters. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2013*.

[11] J. P. Achara, J. Lefruit, V. Roca, and C. Castelluccia. Detecting privacy leaks in the RATP app: How we proceeded and what we found. *Journal of Computer Virology and Hacking Techniques*, 10(4):229–238, 2014.

[12] C. M. Arranz. IP Telephony: Peer-to-peer versus SIP. *MS Thesis, KTH*, 2005.

[13] P. Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, 2010.

[14] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Net-*

*work and Distributed System Security Symposium, NDSS, 2011*.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[16] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the 24th World Wide Web Conference*, 2015.

[17] C. Eubank, M. Melara, D. Perez-Botero, and A. Narayanan. Shining the floodlights on mobile web tracking — a privacy survey. In *Proceedings of Web 2.0 Security and Privacy (W2SP)*, 2013.

[18] Google. Google's Cookie Matching Protocol. https://developers.google.com/ad-exchange/rtb/cookie-guide. Online. Sep, 2015.

[19] S. Han, J. Jung, and D. Wetherall. A study of third-party tracking by mobile apps in the wild. *Technical Report, UW-CSE-12-03-01*, 2012.

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

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

[22] 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*, 2009.

[23] B. Krishnamurthy and C. E. Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proceedings of the 18th World Wide Web Conference*, 2009.

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

[25] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of Web 2.0 Security and Privacy*, 2012.

[26] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[27] L. Olejnik, T. Minh-Dung, C. Castelluccia, et al. Selling off privacy at auction. In *Proceedings of Network and Distributed System Security Symposium*, 2014.

[28] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, 2004.

[29] Y. Xie, F. Yu, and M. Abadi. De-anonymizing the internet using unreliable ids. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2009.

[30] 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 *Proceedings of Network and Distributed System Security Symposium*, 2012.