DECANTeR: DEteCtion of Anomalous outbouNd HTTP TRaffic by Passive Application Fingerprinting

Riccardo Bortolameotti
University of Twente
r.bortolameotti@utwente.nl

Thijs van Ede
University of Twente
t.s.vanede@gmail.com

Marco Caselli
Siemens AG
marco.caselli@siemens.com

Maarten H. Everts
University of Twente and TNO
maarten.everts@tno.nl

Pieter Hartel
Delft University of Technology
pieter.hartel@tudelft.nl

Rick Hofstede
RedSocks Security B.V.
rick.hofstede@redsocks.nl

Willem Jonker
University of Twente
w.jonker@utwente.nl

Andreas Peter
University of Twente
a.peter@utwente.nl

Abstract

We present DECANTeR, a system to detect anomalous outbound HTTP communication, which *passively* extracts *fingerprints* for each application running on a monitored host. The goal of our system is to detect unknown malware and backdoor communication indicated by unknown fingerprints extracted from a host's network traffic. We evaluate a prototype with realistic data from an international organization and datasets composed of malicious traffic. We show that our system achieves a false positive rate of 0.9% for 441 monitored host machines, an average detection rate of 97.7%, and that it cannot be evaded by malware using simple evasion techniques such as using known browser user agent values. We compare our solution with DUMONT [24], the current state-of-the-art IDS which detects HTTP covert communication channels by focusing on benign HTTP traffic. The results show that DECANTeR outperforms DUMONT in terms of detection rate, false positive rate, and even evasion-resistance. Finally, DECANTeR detects 96.8% of information stealers in our dataset, which shows its potential to detect data exfiltration.

1 INTRODUCTION

The latest Verizon Data Breach Investigation Report [31] has shown once again that enterprises across all industries are victim of cyber-attacks. The majority of these attacks is caused by external actors, who are increasingly using malware to exfiltrate data (e.g., steal credentials), spyware to gather information from the victim, and backdoors to communicate with their victims [31]. This highlights the need to identify anomalous outbound traffic.

Most of current network security devices use malware characteristics to identify malicious communication. Predominantly, this is achieved in two different ways. *Signature-based techniques* rely on a dataset of known malware samples and they extract known patterns that characterize the malware. These techniques can also be automated, generating signatures from clusters of malware [20–22]. The difference compared to classic signatures is that automatically generated signatures are more robust against malware variants, because they encode certain traffic characteristics shared by one or more sets of malware. This technique works because much malware shares pieces of the same code. The second category of techniques is *anomaly-based detection*. The main goal is to create a network traffic model based on a set of features that characterizes a specific threat, such as botnets [4, 10, 11] or web attacks [16]. Most of the related work belongs to a subcategory of anomaly detection, that we refer to as *threat-specific* anomaly detection, because all these works either create models trained with malicious data, or focus on specific threats identifiable by specific patterns.

These techniques are the core part of security tools available today, and they have been successful in identifying infected hosts within networks. However, there are two problems with these techniques. Firstly, classifiers or signatures generated based on a set of malware cannot identify new, unknown malware that does not have commonalities with that set. Secondly, there are certain threats, such as data exfiltration or generic backdoors, that are very hard to model due to a lack of clear patterns. For instance, data exfiltration can be an obfuscated transmission of a database in small chunks within hours, or a cryptographic key pair within a single request. To tackle these issues, researchers have proposed anomaly-based detection approaches that generate models only from the benign

network data of each specific machine [5, 24]. We refer to this category as *host-specific* anomaly detection, which differs from the commonly known term 'host-based' used for techniques that analyze the internal state of a machine. Unfortunately, the existing approaches do not provide strong detection performance in realistic scenarios, because they are either easy to evade, or they do not adapt to the host behavioral changes over time, or they trigger too many false positives (FPs).

In this work, we propose DECANTeR, a system that uses a *passive application fingerprinting* technique to model benign traffic only, and hence not relying on any malicious sample. DECANTeR, for each monitored host, *passively* generates *fingerprints* for each application communicating from the host. The fingerprints are composed of different HTTP request features that describe the network behavior of an application. Our solution uses a *hybrid* approach, since the set of features for fingerprints is *dynamically* adapted to the type of the application, and the content of the features represents *static* patterns extracted from the traffic. The intuition is that hosts are characterized by a set of installed applications. Therefore, if malware runs on the host it may generate new fingerprints that show different patterns from those representing host applications. We have evaluated DECANTeR with different datasets—one of which contained traffic of an international organization—and we have compared the results with the current state of the art solution DUMONT [24], showing a clear improvement in the detection performance.

We have chosen to focus on HTTP traffic, since it is a commonly used protocol by malware [28, 34]. We discuss this choice in more detail in Section 2. The novelty of this work lies in passively modeling the benign behavior by identifying different HTTP-based applications of a host from its network traffic, and use these models to identify anomalous behavior in the host communication. In summary, we make the following contributions:

- We present DECANTeR, a solution to detect outbound anomalous HTTP connections, which is based on a *passive application fingerprinting* technique. Our approach automatically generates fingerprints from network traffic, identifying anomalous communications from the monitored hosts. We also discuss how DECANTeR can adapt to behavioral changes of a host over time.
- We have implemented prototypes of DECANTeR and DUMONT [24] in Python, the current state of the art regarding host-specific anomaly detection. We have evaluated and compared them with different datasets. We show that our approach provides better detection performance and is harder to evade.
- We make publicly available both the dataset of data exfiltration malware samples, that we used in our research, and the implementations of DECANTeR and DUMONT.

2 SYSTEM AND THREAT MODEL

We consider a scenario where an enterprise monitors the network traffic of its hosts by routing all traffic through a network monitor that cannot be bypassed. We assume that the network monitor cannot be compromised by an attacker. This is a common assumption because access to the monitor is assumed to be restricted. We also assume there is a security operator that analyzes the alerts produced by the network monitor. The attacker can, however, infect one or more enterprise monitored hosts with malware.

We assume the malware uses HTTP to communicate over the network. We focus on HTTP traffic mainly for two reasons: 1) a large majority of malware uses HTTP [28, 34], either to communicate with their C&C server or to exfiltrate data, because it can camouflage within benign traffic and avoid detection; 2) many enterprise firewalls implement strict filtering rules that block non-web traffic, which forces malware to avoid customized protocols and to use HTTP or HTTPS. Moreover, many enterprises deploy TLS man-in-the-middle (TLS-MITM) proxies in their network [8]. This makes our approach applicable also to HTTPS-based malware, although it is not an optimal solution due to security and privacy concerns. HTTPS-based malware with certificate pinning capabilities that decides not to communicate if it detects a man-in-the-middle attempt, will fail to run in enterprise networks with similar settings. Lastly, HTTP is also used as a protocol for data exfiltration because large quantities of data blend in with the vast amount of benign HTTP data. This makes the detection of data exfiltration extremely challenging for signature-based approaches, especially if the malware obfuscates the data, in fact HTTP-based data exfiltration is still considered an open problem. We assume that a malware can transform data using any combination of compression, encoding or encryption to hide the content.

3 OUR APPROACH

The intuition of our work is the following: all traffic generated by a specific host is the consequence of network activities produced by a set of applications $A = \{a_1, \ldots, a_n\}$ installed on the host. Each application a_i has specific network characteristics, and it is possible to create a fingerprint F_{a_i} for each application. The network traffic of a host H can then be defined as the union of all application fingerprints: $H = \bigcup_{i=1}^{n} F_{a_i}$. Malware is also an application and is likely to have its own fingerprint. This also holds in case of malicious add-in software (e.g., browser add-ons produce a different fingerprint than the browser itself). Therefore, when the malware infects the host and communicates with the outside world, it should be possible to distinguish its traffic because it should differ from the set of benign applications A installed on that host.

Although the intuition seems straightforward, there are several challenges to address. Traditional fingerprinting solutions create fingerprints from offline and often complete datasets, where an application is dynamically analyzed according to different inputs in order to trigger all possible behavior embedded in it [30]. In our setting this would require an analysis a priori of all existing HTTP clients (not only browsers), which is unrealistic. Therefore, one challenge is to generate fingerprints of applications from live traffic, which is likely to be incomplete (due to limited capture time) and heterogeneous (due to differing messages of the same application over time). Secondly, the system should provide an updating mechanism in case new fingerprints are created owing to new software installed on the host. In this work we address both challenges.

3.1 System Overview

DECANTeR has two different modes: *training* and *testing*. The training mode is a setup phase, where the system for a fixed amount of time passively learns for each host its set of fingerprints. Once the training has finished, the testing phase starts. The testing mode runs systematically every X minutes, where X is a timeout determined in the system setup. During this period requests are grouped, labeled and fingerprints are passively extracted from the network. When the timeout occurs, DECANTeR determines whether any of the newly extracted fingerprints are anomalous or not in comparison with the trained fingerprints.

The *training mode* is divided into two modules: labeling and fingerprint generation. The *labeling* module groups information from HTTP requests in different clusters and labels each cluster with its application type (i.e., browser or background). Then, the *fingerprint generation* creates the fingerprint according to the application type of the cluster and its HTTP requests. The outcome of the training mode is a set of fingerprints for each monitored host. There are two options to avoid malicious data in the training phase: the training is done when the host is in a malware-free state (e.g., just formatted or new); or the training phase excludes all the requests that are labeled as malicious by external threat intelligence.

The *testing mode* has three modules: labeling, fingerprint generation and detection. The first two modules work the same as in training mode. The labeling and the fingerprint generation modules extract the set of fingerprints seen in the network in the last X minutes. Next, the *detection* module verifies through different similarity checks whether these newly extracted fingerprints are anomalous or not.

Our solution focuses on outgoing HTTP requests only, and more specifically, GET and POST requests, as these are most commonly used. However, our method can be easily extended to other request types.

3.2 System Details

In this section we discuss the details of each module of DECANTeR.

3.2.1 Labeling. The labeling module takes as input HTTP requests and clusters them according to their `User-Agent` header field, because we want to isolate request generated from distinct applications. Benign applications often use the `User-Agent` to be recognized by web servers, so there is almost a guarantee that all those requests have been generated by the same application, and therefore it is a very efficient way of aggregating during traffic analysis. Each cluster is then analyzed and a label is assigned to it according to its application type. This module runs for a specific timeout that we call *aggregation time t*. During testing mode, t is a fixed time window of X minutes, while in training mode, t matches the length of the training period. When t ends, each labeled cluster is passed to the next module.

Application Types: Background vs. Browser. We have identified two types of HTTP applications: *background* and *browser*. The *background* type represents those applications which traffic content and destination are not directly influenced by user inputs (e.g., an antivirus update query). These applications have predictable behavior, and show fixed patterns in their communication. They often use the same structure of HTTP headers, the communication is often with the same set of domains, and the size and content of the requests is rather similar. The *browser* type represents web browsers, which generate HTTP traffic whose content is unpredictable and dynamic because it directly depends on both user actions and the specific visited web sites, especially considering the widespread use of dynamic web content.

The Labeling Method. The goal of the labeling method is to distinguish between background and browser application clusters. We achieve this by leveraging the dynamic behavior of browser traffic. For example, when a user visits a website, the browser generates a request for a web page (usually HTML). Once the HTML page has been downloaded, the browser generates additional HTTP requests to retrieve extra information such as images, scripts, CSS and others. This information is needed to properly render the webpage. This behavior is unique to browsers, and it is not present in background applications, therefore it can be used to distinguish clusters from these two application types.

Several researchers have already proposed some solutions [17, 32, 35] to encode the dynamic behavior of a browser into a graph data structure known as *Referrer Graph*. Nonetheless, none of them are directly applicable to our setting. ReSurf [32] and ClickMiner [17] focus on reconstructing the user browsing activities from network traffic into a graph by connecting all requests that have been generated by user input, and consequently discarding all the other requests. The Triggering Relation Graph (TGR) [35] connects requests to user input in order to detect stealthy malware activities (i.e., identified by disconnected nodes). User input is collected by an agent hooking browser functions. The TGR approach cannot be used because it requires access to the host, which is outside of our system model. The ClickMiner approach could be used, but processing all servers responses would be too resource demanding for live traffic analysis. Additionally, DECANTeR needs to identify all requests generated from the browser and not only from the user, as ReSurf and ClickMiner do. For these reasons, we introduce a new approach to generate a *Referrer Graph* (see Appendix A.2), introducing the concept of *head nodes*.

Figure 1 shows an overview of the labeling method. For each request we check based on the `CONTENT-TYPE` header if it accepts HTML, javascript, CSS or flash content. If so, we consider it as a *head node*. A head node is a request that may lead to the generation of other requests. Once we have identified all head nodes in a cluster, we link to them all requests that they have spawned; thereby a graph is created. Requests are linked to head nodes if their `REFERER` or `ORIGIN` domain value matches with the `HOST` value of the head node. Head nodes may spawn other head nodes. If at least one graph is present, the connected nodes are moved to a new cluster, which is labeled as browser. The algorithm is depicted in Algorithm 2 in Appendix A.2. Disconnected nodes are further analyzed by an exfiltration filter (Algorithm 3 in Appendix A.2), because we want to check if there are hidden malicious request exfiltrating data with similar header values to the host's browser. Firstly, the exfiltration filter marks all the disconnected requests that are POST or GET with parameters. Secondly, it looks for repeating requests by aggregating all requests with the same URI (without parameters), and verifying that their header fields and values are similar over time. All requests
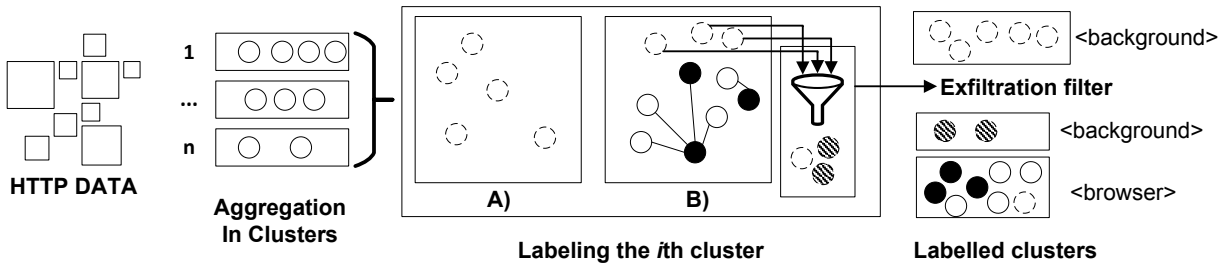
Figure 1: Overview of different cases for the DECANTeR labeling method. A) represents the case of a background application cluster. B) represents the case of a browser application cluster, where two requests are considered suspicious of exfiltrating data and not being a browser. In the figure, each circle represents a request. Black circles are head nodes, outlined circles are requests spawned by head nodes, dashed circles are disconnected nodes, and dashed circles with a pattern are requests that may exfiltrate data.

that are both POST or GET with parameters and repeating over time, are considered as background requests and they are inserted into a new cluster. The other disconnected nodes are inserted in the browser cluster. In case no nodes are connected, we check if the disconnected nodes connect to the graph of the previous time slot $t - 1$. This is needed because requests may be spawned by the head node between the end of one slot, and the beginning of another. If they connect to the previous node, we continue as we discussed above. Otherwise, we label all nodes in the cluster as background. Essentially, this labeling procedure identifies browser and background clusters, and it identifies suspicious background traffic that is hiding within browser traffic.

3.2.2 Fingerprint Generation. This module takes as input the labeled clusters from the labeling module, and for each cluster it generates a fingerprint by extracting a specific set of features, which changes according to the cluster label. We inspect each request in the cluster and we extract the following features:

(1) *Host*: the set of domains that are stored in the HTTP field Host. More specifically, we consider only the top and second level domains. *Intuition*: we have observed that many applications, which often operate as background services, mostly communicate with the same set of domains.

(2) *Constant Header Fields*: the set of header fields *always* present in the cluster requests. *Intuition*: many applications, especially non-browser applications, systematically use a fixed set of header fields for each request they generate, making it a unique characteristic. This feature is not new, but previously it was used to model malicious communication [18], while we use it to model benign software communication.

(3) *Average Size*: the average size of an HTTP request, computed from the sizes of all HTTP requests (including both header and body size). *Intuition*: although the content may vary per request, some applications often generate requests of very similar sizes, especially when they are generated systematically.

(4) *User Agent*: the string of the request field User-Agent. *Intuition*: this value is often unique for each benign application.

(5) *Language*: a set of string values present in the Accept - Language HTTP field. *Intuition*: web browsers use this field to advertise which natural languages they prefer in the HTTP

response. This field characterizes not only the instance of the browser, but also the user settings.

(6) *Outgoing Information*: a close approximation of the total amount of information transmitted by all the requests belonging to the cluster. This feature is used only during testing phase. *Intuition*: we want to keep track of how much information the requests within a cluster have transmitted.

It has been already shown that methods relying on single features, such as the User-Agent string, are not effective [15]. Therefore, we want to create fingerprints that rely on several characteristics of an application network traffic. So, even if the malware makes the right guess of User-Agent, we can still identify the malicious fingerprint as anomalous because the other features may not match with the real application fingerprint.

Outgoing Information. The way the amount of information is calculated is relevant, especially for the case of data exfiltration. If an application periodically generates HTTP requests that always have the exact same content (e.g., antivirus requesting updates), the amount of outgoing information should be the only information contained in the first request. Instead, if an attacker exfiltrates a data item through several HTTP requests, those requests should contain different information, and the amount of outgoing information should be as close as possible to original the item. A naïve approach that quantifies the outgoing information by summing the size of HTTP requests would fail to address these two cases, because it would miss the content differences between requests.

Therefore, we introduce a new method to compute a more precise amount of outgoing information (*OI*). Given a set of requests $\text{REQ}_1 \ldots \text{REQ}_n$ in cluster i,

$$OI_i = \text{size}(\text{REQ}_1) + \sum_{i=2}^{n} \text{LevenshteinDist}(\text{REQ}_i, \text{REQ}_{i-1}).$$

The intuition behind it is that the content of the first request should be considered new information, while the next requests add new information only if they contain new (different) data than their prior request. The full algorithm and its explanation can be found in Appendix A.1.

Different Set of Features for Different Types. The traffic patterns of background and browser applications are almost poles apart. This

diversity is the main reason why we decided to have a different set of features to represent the fingerprints of these two types. Tailoring the set of features according to the characteristics of an application and its type, strengthens the fingerprint against malicious emulation attempts.

We model *background* applications using the following features: *Host* (1), *Constant Header Fields* (2), *Average Size* (3), *User-Agent* (4) and *Outgoing Information* (6). We model *browser* applications by inspecting the *User-Agent* (4), *Language* (5) and *Outgoing Information* (6). These features capture fixed communication patterns that are characteristics of background and browser applications, respectively.

3.2.3 Detection. The detection module takes as input a set of application fingerprints $F_{\text{test}} = \{F_{a_1}, \ldots, F_{a_n}\}$. Each fingerprint F_{a_i} is compared against the fingerprints generated during the training mode $F_{\text{train}} = \{F_{b_1}, \ldots, F_{b_z}\}$. The comparison is done by computing specific similarity functions, which are application-type dependent. In case F_{a_i} is not similar to any of the fingerprints in F_{train}, DECANTeR considers F_{a_i} as a new application. Once a new application is found, DECANTeR verifies if the new fingerprint is a software update (see Section 3.2.4). If F_{a_i} is not an update, an alert is raised if one of these two conditions are satisfied: 1) the amount of outgoing information of F_{a_i} is above a threshold σ, or 2) the user-agent in F_{a_i} resembles a browser user-agent string. 2) is checked by simply verifying if strings such as 'Firefox', 'Chrome', etc. are in the user-agent string.

The detection checks 1) and 2) are used for the following reasons: with 1) we want to know if new applications on the machine are transmitting too much data over the Internet. This may be a sign of a malware installed on the hosts that starts exfiltrating data; with 2) we want to identify those applications that are trying to imitate a browser. This check is based on a common malware behavior, which tries to use user-agent strings of known browsers to hide themselves [23, 34]. Therefore, new browser-looking fingerprints should be considered as anomalies.

3.2.4 Fingerprint Updates. When the detection module finds a new application, it is possible that a false positive is triggered. This can happen for different reasons: a new application has been installed, the application did not communicate during the training phase, or an existing application has been updated. These are events that may happen over time during live monitoring, therefore it is very important that DECANTeR learns from its mistakes.

Common browsers, such as Chrome, often update themselves without user interaction. When a browser is updated, its User-Agent string changes, leading to a new different fingerprint. DECANTeR addresses this issue by verifying for every new fingerprint if the User-Agent string is similar to any of the existing fingerprints. DECANTeR computes the edit distance between the two strings, it divides the outcome with the length of longest string, and it obtains a final value between 0 and 1. If the value is smaller than 0.1, DECANTeR consider the strings to be similar. We have determined this threshold empirically. Therefore, only if a small part of the string changes, they are considered similar (e.g. increased software version). If the strings are similar, DECANTeR runs the similarity functions (again) according to its type, and it automatically assigns the maximum score for similarity function s_4, which is

the similarity function for the user-agent feature (see Appendix B). If the fingerprints are considered similar, the older fingerprint is updated with the new information. The other main causes of false positives are the installation of new software, and software that did not communicate during the training mode. In these cases, DE-CANTeR learns with the help of the security operator. Once he flags an alert as a false positive, DECANTeR can simply add the flagged fingerprint in its pool of trained fingerprints. This method of updating is computationally efficient, because DECANTeR only needs to add an element into a set.

3.2.5 Background Similarity Function. A fingerprint is represented by a set of features. Let us consider F_a and F_b to be two background application fingerprints generated in F_{test} and F_{train}, respectively. F_a and F_b have the same label. The background similarity function verifies whether F_a and F_b are representing the traffic of the same application or not. The function is defined as

$$s_{\text{back}}(F_a, F_b) = \sum_{i=1}^{4} s_i(F_{a_i}, F_{b_i}),$$

where s_i represents a function that checks the similarity of the ith feature (see Section 3.2.2). F_a and F_b are considered similar if and only if $s_{\text{back}}(F_a, F_b) \geq \alpha$, where α is the similarity threshold for s_{back}.

Function s_1 assigns 1 point if *all* the *Host* domains visited by F_a were also seen by F_b, and 0 otherwise. We require all domains to be present because background applications often talk with the same set of domains. If the domain set differs, it might be an indication that the two fingerprints are not representing the same application. Function s_2 assigns 1 point if the *Constant Header Fields* found in F_a traffic matches those in F_b, 0.5 if they are a superset of the headers of F_b, and 0 otherwise. HTTP headers are often repeated in background applications requests. However, sometimes the same application generates requests with additional header fields. This case is addressed giving half of the similarity points. In case any of the header fields observed during training phase is not present, the fingerprints may not represent the same application. Function s_3 assigns 1 point if the absolute difference between F_a and F_b—the *Average Size*—is lower than ϵ, where $\epsilon = \frac{F_{b_3}}{3}$. If it is lower than 2ϵ, 0.5 and 0 otherwise. An exact match is almost impossible to find. Therefore, we use two different intervals that depend on ϵ, which represents an *error rate*, so we address the case where the average size may have changed due to some dynamic properties of the application communication. If the average size is not within the intervals, the fingerprints are likely not generated by the same application. Function s_4 assigns 1 point if the *User-Agent* of the two fingerprints matches, and 0 otherwise. In case there is no match, applications are likely different. An overview of these functions is depicted in Appendix B.

These functions describe patterns that we have observed on real traffic and are tailored to some HTTP characteristics. Through the combination of these four features it is possible to correctly identify the fingerprints sharing the same applications despite small changes in behavior, for example a change in *User-Agent* or communication with a different domain. The 'similarity threshold' α also plays an important role, because it guarantees a certain flexibility. From our empirical evaluation, $\alpha = 2.5$ is the best value

that allows us to match different fingerprints from the same applications, and to distinguish them from those of other applications. Such threshold makes sure that a fingerprint should match with at least three features. The 0.5 is given by a partial match of the constant header features. Lower scores would match fingerprints that do not represent the same application, because they may share the same headers and size, but they communicate with completely different services (e.g., using distinct hosts and user-agents). Higher values do not guarantee flexibility towards changes in the requests. This can be a problem when fingerprints are trained on little data, which contains only a specific subset of the requests generated by the application.

3.2.6 Browser Similarity Function. The browser similarity function is easy to compute, because there are only two features to evaluate:

$$s_{\text{brow}}(F_a, F_b) = s_4(F_{a_4}, F_{b_4}) + s_5(F_{a_5}, F_{b_5})$$

where s_4 is the same as in the background setting. Function s_5 assigns 1 point if the *Language* of the two fingerprints matches, and 0 otherwise.

Fingerprints F_a and F_b are considered similar if and only if $s_{\text{brow}}(F_a, F_b) = \beta$, where β is the similarity threshold for s_{brow}. For browsers both features should exactly match, thus $\beta = 2$. A lower β would result in a more permissive check, allowing one of the two features to not match, which would lead in an easier evasion for the attacker.

4 EVALUATION

In this section we describe the datasets we used to evaluate DE-CANTeR. We discuss how the main system parameters aggregation time t and threshold σ are chosen. We evaluate the detection performance of DECANTeR and compare it with DUMONT [24]. We have implemented a Python version of DECANTeR and DUMONT[1], because its original implementation was not available. In this evaluation, we consider alerts triggered by malware to be true positive, while false positives are those alerts triggered by benign software.

4.1 Datasets

4.1.1 User Dataset (UD). We have collected data from 9 researcher machines at an international university. An overview of the dataset is shown in Appendix C. We used real data to avoid possible biases by capturing data in a lab (e.g., a fixed set of installed applications). The collection of this data is highly privacy sensitive, because it contains all web activities during working hours of the researchers. The period of time of collection varies per user, as it spans from three working days to a few weeks. The dataset contains 123,766 HTTP requests and represents more than 493 hours of network traffic[2].

4.1.2 Organization Dataset (OD). DECANTeR has been deployed to an international organization monitoring outgoing HTTP traffic[3] on a network link with thousands of hosts. The traffic was inspected using Bro [19], which created ad-hoc HTTP logs that have then

been processed by DECANTeR. From the organization, we have obtained 307,053 fingerprints (representing 3,773,106 HTTP requests) generated by DECANTeR for 441 partially self-managed hosts that communicated for a period of a month: 291 employee workstations, and 150 infrastructure machines.

4.1.3 Data Exfiltration Malware (DEM). We analyzed hundreds of malware samples within a virtual machine (VM) for roughly 60 minutes per sample using Cuckoo[4]. In our VM, we have installed known software, stored account credentials for real services (e.g., Gmail, LinkedIn), and placed some decoy documents of different format containing sensitive information, which we obtained from Wikileaks. We have removed all network data samples that generated less than 100 bytes of HTTP traffic. This resulted in 59 traffic malware samples known for their exfiltration capabilities. The samples belong to 8 families of information-stealer malware: iSpy, Shakti, FareIT, CosmicDuke, Ursnif, Pony, Dridex, and SpyEye. The main reason why the number is low is because the (fresh) malware we evaluated was able to detect the VM, and therefore it did not perform any communication. Despite the relatively low number of samples, we believe this dataset can have an important value for the community, because there are two requirements needed to collect it that are not easy to fulfill for all researchers: 1) access to fresh malware samples from specific malware families, because (even few months) old malware may not be able to communicate anymore, and 2) connect the VM to the Internet, which may not be allowed due to risks the infrastructure may incur by running live malware. The collection of this dataset has been approved by the IT department, and the dataset will be public, together with both implementations, to foster research on data exfiltration.

4.1.4 Ransomware (RAN). This dataset consists of ransomware traffic. We obtained 290 pcaps from the authors of FSShield [7][5], and the virtual machine used in their analysis. The VM is used to label benign and malicious traffic (see Section 4.2). We removed those pcaps that did not generated at least 100 bytes of HTTP traffic, obtaining a total of 287 samples. These samples belong to 5 different families of ransomware: CryptoWall, CryptoDefense, Critroni, TeslaCrypt, and Crowti.

4.2 Evaluation Setup

The UD, RAN, and DEM datasets are network traffic files (pcap). Each file is analyzed with Bro [19] to generate a log file, which contains the HTTP headers and additional metadata for each request. The OD dataset is a set of log files.

We have labeled malicious and benign requests in our datasets as follows. In UD, we have manually labeled requests as malicious if they were showing user-agent values not matching any application installed on the machine (e.g., iPhone user-agent on a Windows machine). In RAN and DEM, we have generated network traffic with the VMs (without running malware) and we labeled as benign all requests that had same user-agent as the applications found in the VMs traffic. The other requests are considered malicious, including browser requests because we know that during the analysis only the malware could have used the browser to generate

---

traffic. We have manually labeled the OD dataset after analyzing it with the help of external threat intelligence services[6] and with indicators of compromise we have obtained from a professional threat intelligence provider.

In all experiments with the UD dataset we have used the traffic of the first working day for training and the rest has been used for testing. The training mode—as any setup phase—is trusted, therefore we manually checked if there were malicious fingerprints, and if so we removed them. In the experiments with RAN and DEM we used the VMs traffic generated without malware for training, and all malicious samples are used for testing. In case of the OD dataset, we have trained DECANTeR using the first week of traffic, and the rest was used for testing. In our evaluations, we assume that DECANTeR updates its fingerprints. Thus, if DECANTeR would raise a false positive for five consecutive time slots, and these are identical to each other, only the first is counted as a false positive and the rest is considered true negatives. This represents the real scenario where an operator flags the first alert as a false positive and DECANTeR adds the new fingerprint to its trained set of fingerprints.

4.2.1 Parameter Selection. σ is the system parameter in the detection module that decides whether to trigger an alert or not, depending on the amount of new information generated by a fingerprint. We have evaluated the number of false positives (FP) σ would raise on the UD according to different values. Figure 2 suggests that the number of FPs is proportionally inverse to the threshold value. A high threshold produces few FP. Obviously, a too high threshold can also lead to a low detection of anomalous traffic. We suggest $\sigma = 1000$ bytes, because the decrease of FP is less significant for higher thresholds. Moreover, 1000 bytes is still a low value that could detect an exfiltration of very small but sensitive data items such as cryptographic keys (e.g., a .pem file with a 2048-bit RSA key has a size of 1700 bytes).

Another relevant parameter of DECANTeR is the aggregation time t during the testing mode. The advantages of a long detection time are: the lower number of FP, because more requests are aggregated in the same fingerprint; and an attacker, who wants to remain undetected, must severely decrease its communication. If we set $t = 30$ minutes and $\sigma = 1000$ bytes it means that the attacker cannot transmit more than 1000 bytes within half an hour. The disadvantage is slow detection, because the detection is performed less often and an attacker can transmit more data before it is detected. A low detection time has the opposite advantages and disadvantages. We have tested DECANTeR with $\sigma = 1000$ and three different t values to understand the relation between t and the number of FP triggered. We used $t = 1$, 10 and 30 minutes. The overall number of FPs DECANTeR triggered was 116, 73, and 63 respectively. Showing a clear relation between time and FP reduction. Both 10 and 30 minutes cases give an acceptable number of FP, however we consider 10 minutes a better tradeoff because it still provides enough time to the operator to verify the alert, and it provides quicker detection. If having low FPs is a necessity, 30 or more minutes as t values is a better option. For the rest of the paper, we use $t = 10$ minutes and $\sigma = 1000$ bytes.
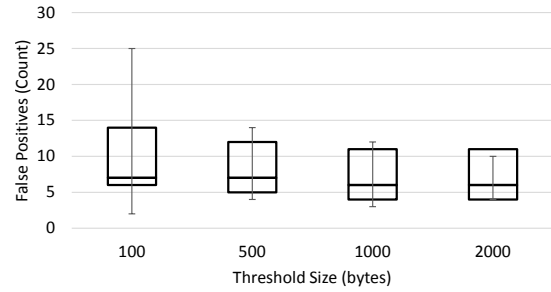

Figure 2: Represents the number of false positives generated from the traffic of 9 users (i.e., UD dataset) for different values of σ, and aggregation time $t = 10$ minutes. Error bars show the outliers.

4.2.2 Detection Performance. We have evaluated the detection performance of DECANTeR against the UD, RAN, and DEM datasets. We have trained and tested DECANTeR for each dataset as discussed in Section 4.2. The analysis evaluates the number of fingerprints that are correctly classified (or not) by DECANTeR. Fingerprints are labeled as malicious if they contain at least one request previously labeled as malicious, otherwise they are considered benign. The results are shown in Table 1.

In the UD dataset, DECANTeR detected 117 malicious fingerprints for a specific user despite the use of a known antivirus software. The user was infected by an adware, which explains the presence of many different fingerprints. Many requests contained distinct user-agent values (even mobile strings), header fields and domains. Possibly, the malware wanted to emulate various requests, as they were generated from distinct hosts, to increase the number of visits (or clicks) to specific ads links. Although adware may not seem very harmful, we know that they also have data exfiltration capabilities [29]. The FNs were all unknown fingerprints, meaning that they did not match any of the existing fingerprints. However, DECANTeR did not trigger an alert because they transmitted fewer bytes than the threshold σ (i.e., 1000 bytes) during the aggregation period (i.e., in our evaluation is 10 minutes). Additionally, those fingerprints did not have a user-agent value of a known browser, which is the last check that triggers alerts in case of fingerprints that transmit little data. In Section 5 we explain additional evasion techniques that malware can use against DECANTeR, despite we did not encounter many of them in our analyses. The FPs were mainly caused by applications that were not fingerprinted during training mode. For instance, traffic generated by a Windows VM on a Linux host. Other FPs were triggered due to labeling 'mistakes', where some browser requests were considered as background. This mainly happened in case of web scripts, which do not reference to previous requests and send out information through GET with parameters (or cookies), or OSCP POST requests. Nonetheless, when DECANTeR is updated, it learns the background fingerprints of these scripts, and next time they are triggered DECANTeR will not raise a FP. Since we do not know the amount of malicious software running on the infected host, we do not know the number of malicious samples DECANTeR has detected.

Table 1: Detection performance for different datasets with $\sigma = 1000$ and $t = 10$ minutes. Malware Detection indicates the percentage of detected samples by DECANTeR. For the classification performance, the values represent the number of fingerprints classified as true positives (TP), false negatives (FN), true negatives (TN) or false positives (FP). FPR indicates the false positive rate $FPR = \frac{FP}{FP+TN}$, while TPR indicates the true positive rate $TPR = \frac{TP}{TP+FN}$.

Dataset	Malware Detection	Classification Performance					
		TP	FN	TN	FP	FPR	TPR
UD	-	117	36	4291	73	1.6%	76.4%
RAN	98.6%	3348	438	4257	2	0%	88.4%
DEM	96.8%	237	67	24	1	4%	77.9%

In RAN and DEM datasets, DECANTeR classifies most of the fingerprints correctly. DECANTeR on average detects 8 malicious fingerprints out of 10 (see Table 1). Considering that a malicious sample is successfully detected if at least one TP is triggered to the operator, our system detected 98.6% and 96.8% of malicious sample from the RAN and DEM dataset, respectively. The FPR for the DEM dataset seems very high, but only 1 FP has been in fact triggered by DECANTeR.

Similarly to the UD dataset, FNs are mostly caused by malware that have fingerprints with low outgoing information. Moreover, a few malware samples (i.e., Ursnif family[7]), which were successfully detected, generated additional browser-like traffic that was not properly classified. The samples in question turned out to generate traffic that matched the browser traffic of our VM, because it generated a Referrer Graph and it has been analyzed accordingly by DECANTeR. Therefore, the communication was considered as a FN. However, this was only noise created by the malware, which communicated with its C&C through other requests that had different user agents and were not connected in any graph. This led to the generation of background fingerprints that have been then triggered by DECANTeR as anomalous, resulting in a successful detection of the sample. Another technique used by other Ursnif samples, was to exfiltrate data using 'Microsoft Crypto API 6.3' as user-agent and exfiltrated the data (3000 bytes per request) through a header field. This seemed to be a mimicking attempt. DECANTeR detected this case as well, because the OS fingerprint for the crypto API was version 6.1, the size of the fingerprints was around few hundred bytes instead of thousands as for Ursnif, and the constant headers were different.

4.2.3 Comparison with DUMONT. The closest *host-specific* related work to DECANTeR are WebTap[5] and DUMONT [24]. In [24] the authors have already compared these two solutions, showing that WebTap suffers of a high false positive rate, and the detection rate drops significantly in case malware would use simple evasion techniques. Therefore, we compare DECANTeR with DUMONT.

DUMONT creates a one-class SVM from HTTP requests according to 17 different numerical features that involve different metrics

such as entropy, length of header fields, and temporal traffic characteristics. During a testing phase, DUMONT verifies if the features of each new requests are within a certain distance from the sphere represented by the one-class SVM. Since the data used to evaluate DUMONT was not available, and nor it was the implementation, we have implemented it ourselves and we used our datasets for the comparison. Our implementations takes as input only HTTP headers, therefore DUMONT cannot use the entropy features for HTTP POST. Considering that the number of POSTs in the network are by far lower than GET requests, we do not consider this a relevant issue. For each experiment, we have followed the procedure discussed in [24], by calibrating the model with malicious data (from both RAN and DEM datasets), and tested it with different parameters (i.e., one-class SVM soft margin) to find a suitable ratio of false positive and detection rate. In our results, we discuss two different values used to compute the optimal soft margin: 0.1 that triggers a low number of FPs, and 0.6 that gives a reasonable ratio between FPs and TPs.

For a fair comparison we have evaluated the correct classification of requests, in contrast to fingerprints since DUMONT works only on requests. We considered for DECANTeR all requests in malicious fingerprints to be malicious, and similarly for those labeled as benign. After all, fingerprints are abstractions of a group of requests. The results are shown in Table 2. The first observation is that the detection performance of DECANTeR are better than those shown in Table 1. The different results between the two evaluations are related to the distribution of requests across the fingerprints. For example, for two fingerprints, one classified as TP containing 5 requests, and one FN with 1 request, the TPR becomes 0.83, while if we consider only the fingerprints the TPR is 0.5. The second observation is that DECANTeR clearly outperforms DUMONT in all three detection aspects: FPR, TPR and Malware Detection (as shown in the tables). One of the biggest differences lies in the detection of malicious sample, where DECANTeR shows consistent detection independently from the underlying malicious data, while DUMONT detection strongly suffer this depedency.

The low detection performance of DUMONT is explainable by the fact that many malicious requests are not structured (e.g., length of header fields) much differently than benign requests, therefore it likely misses those requests. DECANTeR overcomes this issue by using semantic information and different features per application type, leading to a higher specificity. A surprising result is the high number of FPs in the UD dataset triggered by DUMONT. This may be caused by two different behaviors. Firstly, the training set did not contained all possible applications, so requests generated in testing phase by VMs, new browsers, and other applications may have influenced DUMONT. Secondly, many benign requests contain big amount of data (e.g., COOKIE), which are bigger than average HTTP requests. DECANTeR deals with these behaviors by adapting over time through the update mechanism, and by modeling the traffic according to its application type.

4.2.4 OD Dataset Analysis. DECANTeR detected 8 machines with actual anomalous behavior. Half of these machines showed traffic patterns known to be caused by malware according to different sources. The other half has shown known anomalous patterns related to advertisement websites, which suggests the presence of

---

[7]https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32/Ursnif

Table 2: Comparison between DECANTeR and DUMONT, evaluated over our three datasets. We show the results for DUMONT according to two different threshold, one (0.1) that is conservative and tries to raise the least number of FPs, and the other (0.6) that shows a better ratio FPs and TPs.

System	Dataset	Malware Detection	Classification Performance					
			TP	FN	TN	FP	FPR	TPR
DECANTeR	UD	-	928	51	90566	3378	3.5%	94.7%
	RAN	98.6%	81910	1123	13520	10	0%	98.6%
	DEM	96.8%	4887	2643	352	3	0.8%	65%
DUMONT .1	UD	-	49	930	87003	6959	7.4%	5%
	RAN	81.8%	17426	65607	13529	4	0%	20%
	DEM	4%	20	7513	351	4	1%	0.2%
DUMONT .6	UD	-	164	815	64824	29138	31%	16.7%
	RAN	100%	81708	1325	1203	12330	91.1%	98.4%
	DEM	40.5%	2688	4845	132	223	62.8%	35.6%

Table 3: Evasion test against DECANTeR and DUMONT

System	Dataset	TPR	Malware Detection
DECANTeR (requests)	RAN	99.9%	100%
	DEM	99.2%	100%
DUMONT .1	RAN	4.3%	25.6%
	DEM	0.4%	5.5%
DUMONT .6	RAN	58.4%	100%
	DEM	15.8%	62.1%

some type of adware, perhaps browser haijackers. Unfortunately, we could not check directly the host machine to get further proofs. In one case, DECANTeR has identified a malicious IP address before being blacklisted by VirusTotal. Overall the FPR[8] is 0.9%. The specific FPR values per each machine category are: 1% for workstations and 0.3% for servers. These values are expected, because workstations produce much more outgoing HTTP traffic.

4.2.5 The Evasion Test. As we know malware tries to imitate browser user-agents strings [23, 34]. A malware can choose an existing and valid browser user-agent or even copy the same as its victim's browser, by inspecting the OS (e.g., Windows Registry) or by sniffing the network. We evaluated how DECANTeR performs in case malware would use the same user-agent as its victim, and even its language. A similar test was performed in [24]. We give the malware the exact features needed to bypass our browser similarity check. We have modified all malicious requests in the DEM and RAN logs by substituting the original user-agent value with the one of the VM browser, which in this case was the victim. We did the same for the accept-language header, and if it was not present we have injected it in the log. We have tested both DECANTeR and DUMONT, and again for fair comparison we considered the requests. The results shown in Table 3 ironically shows an increase in detection for DECANTeR, even though one may expect a drop of detection. The reason lies in the labeling method. Although part of the content of the message is exactly the same as the real browser, malicious requests are still labeled as background, because they do not create a referrer graph. Moreover, since they all share the same user agent, their amount of outgoing information adds up and it always exceeds σ. The results also show that DECANTeR is more robust than DUMONT against these simple evasion attempts.

5 EVASION
The evasion test has shown that DECANTeR is not easy to evade with simple evasion techniques such as spoofing user agent or other header values. However, this does not make DECANTeR impossible to evade.

The first type of evasion is to exploit σ and t, by communicating little data in each time slot without triggering the threshold,

and using a non-browser user-agent string. This type evasion is inherent in anomaly detection, because it relates to the thresholds and timeouts used for detection. It still possible to reduce the risk of FNs by reducing σ and t, however this could lead to a larger number of FPs as well. Lastly, randomizing σ and t within a certain range of values can make the system less predictable to the attacker, and therefore may lead him to make mistakes that can cost him the detection. Nonetheless, the attacker can still evade using the lowest values of σ and t.

A malware can evade DECANTeR by mimicking the dynamic behavior of a browser and by modifying its user-agent and language strings as the victim's browser. These changes require malware to evolve from the simple techniques they use today. If any of these two conditions fails, the malware is likely to be detected either because labeled as a background application, or because it would generate a different fingerprint than the victim browsers. An example are the Ursnif samples, where the malware generated browser traffic from Firefox installed on the VM only to create noise and hide its real C&C communication. However, the real malicious communication did not show any dynamic pattern and did not have any shared characteristics with the VM browser. Therefore, the communication has been successfully detected despite the noise.

Another way of evading DECANTeR is to emulate an installed background application. However, assuming the malware can spoof the correct user agent and host values for the target application, this evasion has two main disadvantages: 1) the malicious client should adapt the constant headers of its request, which may create compatibility issues with the server implementation; 2) the average size of background application is often small, therefore to avoid detection malware should also generate small requests. This slows down possible data exfiltration, even though HTTP is used for its capabilities of transfer a lot of data in short amount of time. An example is again Ursnif samples that tried to emulate the HTTP requests of the Microsoft Crypto API, but the headers, the average size and the user-agents of the requests did not match the characteristics of the Microsoft Crypto API version of the victim, therefore it has been successfully detected.

A more advanced technique is to create a request with Referrer field matching a head node request generated by the real browser. In this case the malicious request would be connected to the graph and it would be white-listed. This attack would bypass the main mechanism of distinction between background and browser applications, allowing the attacker to exfiltrate the data while camouflaging as browser. However, this attack is quite advanced since the malware should be able to monitor the network and to adapt its content according to live traffic details.

---

[8]Also in this scenario we assume an operator is updating DECANTeR.

Lastly, the attacker can try to *poison* the fingerprint update mechanism by convincing DECANTeR to update a benign fingerprint with a malicious one. For example, the malware can start generating HTTP requests with a user-agent similar to the victim browser, and same language feature. This would trigger an update, and future connections of the malware will not be detected. However, DECANTeR can detect this by monitoring the old user-agent string (before the update take place). In case DECANTeR detects requests with the old user-agent, then a poisoning attack has been detected. This works because, once it updates, benign software does not switch back to the old value.

6 DISCUSSION

6.1 Fingerprinting Technique

A central component in DECANTeR is the Referrer Graph, which tries to abstract the browser dynamics to distinguish between background and browser applications by leveraging the requests generated by browsers to download the website resources.

There are two situations that might be problematic for DECANTeR, but in practice they are infrequent.

The first is when websites do not require further resources other than the HTML itself, in which case a graph does not exist for these requests. In case the browser has accessed only this 'type' of websites within t, DECANTeR would generate a background fingerprint, and it would trigger an alert (i.e., FP) because, despite the low outgoing information, the fingerprint has a known browser user-agent. However, it is more likely that, within t, the browser accesses also other websites with additional resources to download. In this case, there would be at least one graph for the cluster, and a browser label would be assigned to it. The disconnected nodes (e.g., the requests to websites without extra downloadable content) would be checked against the exfiltration filter (see Algorithm 3) as discussed in Section 3.2.3. These requests probably do not show signs of exfiltration, therefore they are just assigned to the cluster previously labeled as browser.

The second problematic situation *may* happen when non-browser applications use the REFERER field (e.g., cloud storage or chat clients). Do note that we did not find any example of such an application in our analyses, probably because they were encapsulated in TLS and we did not use a TLS proxy. In such cases a Referrer Graph would be present and the application would be labeled as a browser, but it will likely have a different fingerprint because, for example, the user-agent will differ from the host browsers. This should not pose a problem as the fingerprint would also be create in the training phase, or updated after the first FP, which means that a tested application is matched with this fingerprint. In case a browser is used to upload data on a cloud storage service (e.g., Google Drive) or a messaging service (e.g., Facebook Chat), DECANTeR generates fingerprint that matches the browser fingerprint. This happens because the presence of REFERER and ORIGIN fields generate a Referrer Graph for the cluster, and user-agent and language match those of the browser. This can be checked using a tool that intercept HTTP/HTTPS requests, such as Burp Suite[9].

[9]https://portswigger.net/burp/

6.2 Passive Application Fingerprinting

The detection evaluation has shown that DECANTeR is capable of detecting malicious communication with high success rate, despite being trained *without* malicious sample, while at the same time producing a FPR of 1% and 1.6% on the OD (see Section 4.2.4) and UD datasets (see Section 4.2.2), respectively. These results are a consequence of the passive application fingerprinting technique (PAF). Fingerprints are abstract representations of web requests and they encode some semantic information about the connections. These abstractions allow DECANTeR to classify sets of requests correctly despite possible changes in their structure or content, which are quite common in heterogeneous protocols such as HTTP. Solutions such as DUMONT suffer such changes. An additional benefit of PAF is the *simple* and *intuitive* mechanism it provides for updating. This is essential in host-specific approaches, where traffic may change over time.

DECANTeR can be considered as a significant step to bring host-specific anomaly-based detection into practice. The results have shown a great improvement with respect to the current state of the art, and they have shown this approach can be practical. At the current stage, we believe the best use of DECANTeR is to monitor a subset of hosts, especially those that are known to store sensitive data or to perform sensitive activities (e.g., board members workstations, admins). DECANTeR can be improved in many ways. For example, we could improve the labeling method or the way fingerprints are compared and generated, which are still in an experimental stage. Another idea would be to cross-check background with browser profiles (or viceversa), but this would also increase the risk of FNs, therefore we do not consider it as a good option.

6.3 Use-case: Data Exfiltration

DECANTeR has detected 96.8% of data exfiltration malware and also detected a tool specialized in data exfiltration (i.e., Data Exfiltration Toolkit)[10]. We believe DECANTeR is a good fit for detecting data exfiltration because the detection works indepedently from the content of the communication (i.e., payload), which is often obfuscated by attackers (e.g., using steganography [36]) as main mechanism to avoid detection over the network.

From the network perspective it is impossible to determine whether a specific communication contains sensitive data when it is obfuscated. This is the main reason why current solutions [2, 12, 14, 25–27] fail, as they try to identify and stop sensitive data when it is transmitted over the network, but data has been already obfuscated and it cannot be identified. Approaches that try to detect anomalous encrypted outbound communication [1, 13] also fail, because they rely on the high entropy values of encryption or compression. However, when malware uses encoding after encryption, the entropy drastically drops and the exfiltration is not detected. An approach that quantifies the amount of leaked information seems more appropriate [6]. DECANTeR combines a leakage quantification approach with application fingerprinting to detect new software exfiltrating data. This combination has shown good performance by detecting 96.8% of info stealers samples, where the current state of the art host-specific approach reached only 40%.

[10]https://github.com/sensepost/DET

7 RELATED WORK

In this section we discuss the related work regarding threat-specific and host-specific approaches.

Threat-Specific Approaches. *Automated Signature Generation from Malware Clusters*: researchers have proposed several approaches to cluster malware samples according to network features, and generate signatures from these clusters [18, 20–22]. Rafique and Caballero proposed FIRMA [22], a tool that aggregates malware samples into families based on similar protocol (e.g., HTTP, SMTP, and IRC) features and generates a set of network signatures for each family. Perdisci et al. [20, 21] proposed a technique that clusters malware according to URL similarities, and from the URL it extracts subtokens that are used to identify malicious communications on the network. Nelms et al. [18] present a technique based on adaptive templates that are created from observations of known botnet traffic, which can be used to detect bots on live networks and even identify to what family they belong. Zand et al. [33] propose a method to generate signatures by identifying and ranking the most relevant and frequent strings in malicious traffic. Zarras et al. [34] propose BOTHOUND, a system that extracts all header chains (i.e., set of header fields in an HTTP request) for benign and malicious software, and malicious requests are identified if their header chains are different than known benign software, or if they match, but their HTTP template matches an existing malicious template. DECANTeR differs from these techniques because it does not create signatures from sets of known malware samples.

Anomaly-based Threat-specific Detection: several research studies have explored anomaly-based techniques to identify botnet traffic in a network [4, 10, 11]. These techniques leverage specific network patterns shown by botnets and use malicious samples to train their detection models. For instance, multiple hosts within a network infected by the same bot have common communication patterns. Bartos et al. [3] build a classifier that recognizes malicious behavior and is optimized to be invariant to malware behavioral changes. However, this approach also requires malware during the training of the classifier. Many other studies have proposed anomaly-based detection techniques that analyze different features that can represent the behavior of a specific threat. It is applied in the context of web-attacks [16], distributed denial of services [9], encrypted data exfiltration [13], and others. Although we also use an anomaly-based detection approach, we use neither particular features to detect specific threats, nor known malicious samples to train our models. DECANTeR focuses on modeling *only* benign behavior, and it identifies malicious traffic by observing anomalies.

Host-Specific Approaches. This category contains approaches that model only the normal network behavior generated by a host, without additional knowledge of threats or known malware samples. In [35], Zhang et al. propose a user-intention based approach to detect communication of stealth malware. The proposed method monitors network and host activities, and it creates a triggering relation graph (TRG), a graph that binds a set of HTTP requests to the user action that has triggered them. This approach detects malware because its HTTP traffic does not correlate with user activities. However, it uses host-based information (e.g., process id), which is outside our system model. WebTap [5] is a tool that creates a statistical model of the browsing behavior of a user according to

different features (e.g., header information, bandwidth, request size, and request regularity). WebTap uses this information to identify unknown HTTP traffic and to detect covert communication. However, it has a high false positive rate of 12% because the tool only models browser traffic, so background traffic is treated as anomalous. This makes WebTap not practical. Schwenk and Rieck proposed DUMONT [24], a system to detect covert communication over HTTP. The general approach is similar to WebTap, however they use a one-class SVM classifier to build the HTTP-traffic model of users. DUMONT uses several numerical features of HTTP headers only, with the ultimate goal of characterizing the 'average' HTTP request for each use. The detection performance is worse (89.3% average detection rate) than WebTap, however DUMONT generates a much lower number of false positives, and it suffers less simple evasion attempts. DECANTeR differs from WebTap and DUMONT because it models benign traffic with a new technique: passive application fingerprinting. Moreover, in contrast with WebTap and DUMONT, DECANTeR provides some mechanisms to adapt fingerprints to changes of hosts behavior.

8 CONCLUSION

In this work we have shown how HTTP-based applications can be fingerprinted and used to detect anomalous communications. This technique can be used *without* using malicious data during the training phase, therefore it avoids any possible bias from specific malware samples. Moreover, the proposed technique detects anomalous communication independently from their payload, thereby being a promising solution for data exfiltration and unknown malware. This distinguishes our work from most of the existing solutions, which often model network traffic to detect specific attacks or malware behavior (extracted from clusters of known malware), or tries to identify sensitive data within the network payload. We have implemented this technique in a system called DECANTeR, and we have evaluated it, showing better detection performance than other state of the art solutions.

REFERENCES

[1] Areej Al-Bataineh and Gregory White. 2012. Analysis and detection of malicious data exfiltration in web traffic. In *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*. IEEE Computer Society, 26–31.

[2] Sultan Alneyadi, Elankayer Sithirasenan, and Vallipuram Muthukkumarasamy. 2015. Detecting Data Semantic: A Data Leakage Prevention Approach. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. IEEE, 910–917.

[3] Karel Bartos, Michal Sofka, and Vojtech Franc. 2016. Optimized Invariant Representation of Network Traffic for Detecting Unseen Malware Variants. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. USENIX Association, 807–822.

[4] Leyla Bilge, Davide Balzarotti, William K. Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: Detecting Botnet Command and Control Servers through Large-Scale NetFlow Analysis. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*. ACM, 129–138.

[5] Kevin Borders and Atul Prakash. 2004. Web Tap: Detecting Covert Web Traffic. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004.* ACM, 110–120.

[6] Kevin Borders and Atul Prakash. 2009. Quantifying Information Leaks in Outbound Web Traffic. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA.* IEEE Computer Society, 129–140.

[7] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. ShieldFS: a Self-Healing, Ransomware-aware Filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016.* ACM, 336–347.

[8] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Micheal Bailey, J. Alex Halderman, and Vern Paxson. 2017. The Security Impact of HTTPS Interception. In *Proceedings of the Network and Distributed System Security Symposium, NDSS, San Diego, California, USA, 26th February - 1st March, 2017.* The Internet Society.

[9] Laura Feinstein, Dan Schnackenberg, Ravindra Balupari, and Darrell Kindred. 2003. Statistical Approaches to DDoS Attack Detection and Response. In *3rd DARPA Information Survivability Conference and Exposition (DISCEX-III 2003), 22-24 April 2003, Washington, DC, USA.* IEEE Computer Society, 303.

[10] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. 2008. BotMiner: Clustering Analysis of Network Traffic for Protocol and Structure-Independent Botnet Detection. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA.* USENIX Association, 139–154.

[11] Guofei Gu, Junjie Zhang, and Wenke Lee. 2008. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, CA, USA, 10th February - 13th February 2008.* The Internet Society.

[12] Michael Hart, Pratyusa K. Manadhata, and Rob Johnson. 2011. Text Classification for Data Loss Prevention. In *Privacy Enhancing Technologies - 11th International Symposium, PETS 2011, Waterloo, ON, Canada, July 27-29, 2011. Proceedings.* Springer, 18–37.

[13] Gaofeng He, Tao Zhang, Yuanyuan Ma, and Bingfeng Xu. 2014. A Novel Method to Detect Encrypted Data Exfiltration. In *Second International Conference on Advanced Cloud and Big Data, CBD 2014, Huangshan, China, November 20-22, 2014.* IEEE Computer Society, 240–246.

[14] Gilad Katz, Yuval Elovici, and Bracha Shapira. 2014. CoBAn: A Context-based Model for Data Leakage Prevention. *Information Sciences* 262 (2014), 137–158.

[15] Nizar Kheir. 2012. Analyzing HTTP User Agent Anomalies for Malware Detection. In *Data Privacy Management and Autonomous Spontaneous Security, 7th International Workshop, DPM 2012, Pisa, Italy, September 13-14, 2012.* Springer, 187–200.

[16] Christopher Krügel and Giovanni Vigna. 2003. Anomaly Detection of Web-based Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washington, DC, USA, October 27-30, 2003.* ACM, 251–261.

[17] Christopher Neasbitt, Roberto Perdisci, Kang Li, and Terry Nelms. 2014. Click-Miner: Towards Forensic Reconstruction of User-Browser Interactions from Network Traces. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014.* 1244–1255.

[18] Terry Nelms, Roberto Perdisci, and Mustaque Ahamad. 2013. ExecScent: Mining for New C&C Domains in Live Networks with Adaptive Control Protocol Templates. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013.* USENIX Association, 589–604.

[19] Vern Paxson. 1999. Bro: a System for Detecting Network Intruders in Real-time. *Computer Networks* 31, 23-24 (1999), 2435–2463.

[20] Roberto Perdisci, Davide Ariu, and Giorgio Giacinto. 2013. Scalable Fine-grained Behavioral Clustering of HTTP-based Malware. *Computer Networks* 57, 2 (2013), 487–500.

[21] Roberto Perdisci, Wenke Lee, and Nick Feamster. 2010. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA.* USENIX Association, 391–404.

[22] M. Zubair Rafique and Juan Caballero. 2013. FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors. In *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings.* Springer, 144–163.

[23] Christian Rossow, Christian J Dietrich, Herbert Bos, Lorenzo Cavallaro, Maarten Van Steen, Felix C Freiling, and Norbert Pohlmann. 2011. Sandnet: Network traffic analysis of malicious software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security.* ACM, 78–88.

[24] Guido Schwenk and Konrad Rieck. 2011. Adaptive Detection of Covert Communication in HTTP Requests. In *Seventh European Conference on Computer Network Defense, EC2ND 2011, Gothenburg, Sweden, September 6-7, 2011.* IEEE Computer Society, 25–32.

[25] Xiaokui Shu and Danfeng Yao. 2012. Data Leak Detection as a Service. In *Security and Privacy in Communication Networks - 8th International ICST Conference, SecureComm 2012, Padua, Italy, September 3-5, 2012.* Springer, 222–240.

[26] Xiaokui Shu, Danfeng Yao, and Elisa Bertino. 2015. Privacy-Preserving Detection of Sensitive Data Exposure. *IEEE Trans. Information Forensics and Security* 10, 5 (2015), 1092–1103.

[27] Xiaokui Shu, Jing Zhang, Danfeng Yao, and Wu-chun Feng. 2016. Fast Detection of Transformed Data Leaks. *IEEE Trans. Information Forensics and Security* 11, 3 (2016), 528–542.

[28] Aditya K. Sood, Sherali Zeadally, and Richard J. Enbody. 2016. An Empirical Study of HTTP-based Financial Botnets. *IEEE Trans. Dependable Sec. Comput.* 13, 2 (2016), 236–251.

[29] Veronica Valeros. 2016. In plain sight: Credential and data stealing adware. (2016). http://blogs.cisco.com/security/in-plain-sight-credential-and-data-stealing-adware

[30] Shobha Venkataraman, Juan Caballero, Pongsin Poosankam, Min Gyung Kang, and Dawn Xiaodong Song. 2007. Fig: Automatic Fingerprint Generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007.* The Internet Society.

[31] Verizon. 2016. Data Breach Investigations Report. (2016). http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/

[32] Guowu Xie, Marios Iliofotou, Thomas Karagiannis, Michalis Faloutsos, and Yaohui Jin. 2013. ReSurf: Reconstructing Web-Surfing Activity from Network Traffic. In *IFIP Networking Conference, 2013, Brooklyn, New York, USA, 22-24 May, 2013.* IEEE, 1–9.

[33] Ali Zand, Giovanni Vigna, Xifeng Yan, and Christopher Kruegel. 2014. Extracting Probable Command and Control Signatures for Detecting Botnets. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014.* 1657–1662.

[34] Apostolis Zarras, Antonis Papadogiannakis, Robert Gawlik, and Thorsten Holz. 2014. Automated Generation of Models for Fast and Precise Detection of HTTP-based Malware. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014.* 249–256.

[35] Hao Zhang, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Zhibin Zhang. 2016. Causality Reasoning about Network Events for Detecting Stealthy Malware Activities. *Computers & Security* 58 (2016), 180–198.

[36] Elzbieta Zielinska, Wojciech Mazurczyk, and Krzysztof Szczypiorski. 2014. Trends in Steganography. *Commununications of ACM* 57, 3 (2014), 86–95.

A ALGORITHMS

A.1 Outgoing Information

Algorithm 1 Computation of Outgoing Information

Require: the current request REQ_i, the latest request REQ_{i-1} transmitted by the same application.

1: **procedure** COMPUTEOUTINFO(REQ_i, REQ_{i-1})
2:     $OI \leftarrow 0$ ▷ Initialize counter
3:     **for all** h_j in REQ_i **do**   ▷ h_j is the jth header in REQ
4:         **if** h_j in REQ_{i-1} **then**
5:             OI += edit_dist ($REQ_i.h_j$, $REQ_{i-1}.h_j$)
6:         **else**
7:             OI += length($REQ_i.h_j$)
8:         **end if**
9:     **end for**
10:     OI += edit_dist(REQ_i.body, REQ_{i-1}.body)
11:     updateCache($REQ_i.h_{\text{user-agent}}$, REQ_i)
12: **return** OI
13: **end procedure**

The outgoing information is computed through ComputeOutInfo, which takes as input two values: the new incoming request REQ_i, and the latest analyzed request REQ_{i-1}, which is stored in a cache (i.e., hash table) that uses the user-agent strings as keys. The function verifies if there are headers (including URI) present in REQ_i that are also present in REQ_{i-1} (line 3-4). If so, it computes the edit

distance (i.e., Levenshtein distance) between the two header values. The result is the new information introduced by the specific header value of REQ_i and it is added to the counter of outgoing information (line 5). For those headers that are present in REQ_i but not in REQ_{i-1}, we consider their values as new information (line 7). Then the edit distance between the body of the requests is computed and is added to OI (line 10). Once all headers have been analyzed, we update the cache by substituting REQ_{i-1} with REQ_i (line 11), because now REQ_i is the last analyzed request.

A.2 Labeling Algorithms

Algorithm 2 Graph creation

1: **procedure** CREATEGRAPH(*requests*)
2:     $graph \leftarrow \{\}$
3:     $headNodes \leftarrow []$
4:     **for** r in *requests* **do**
5:         **if** r.content-type == HTML or
6:             r.content-type == CSS or
7:             r.content-type == JavaScript or
8:             r.content-type == Flash **then**
9:                 $headNodes \leftarrow r$
10:         **end if**
11:     **end for**
12:     **for** r in *requests* **do**
13:         $graph$.addNode(r)
14:         **for** *head* in reversed(*headNodes*) **do**
15:             **if** r.referrer == *head*.host **then**
16:                 $graph$.addEdge(*head*, r)
17:                 **break**
18:             **end if**
19:         **end for**
20:     **end for**
        return *graph*
21: **end procedure**

The labeling method considers a cluster of HTTP requests to be a browser-type application if there is at least one Referrer Graph connecting the requests. The Algorithm 2 describes how a Referrer Graph is built from a set of requests. For each request, we check if in its CONTENT-TYPE header has a known value (i.e., html, css, js, flash). If so, that request is considered a head node. The second step is to iterate over the requests once again, and verify if each request is linked with a head node. A request is linked to a head node if its REFERER or ORIGIN header value matches with the HOST header value of a head node. A graph is created if at least one request is connected to a head node.

All requests that are not connected into a graph are represented by disconnected nodes. These nodes are further analyzed to detect potential data exfiltration. For instance, a malware can have the same user-agent of the victim's browser, and it exfiltrates data through some HTTP requests. To identify these potential anomalies we inspect disconnected nodes with an exfiltration filter. Algorithm 3 describes the details of the filter. First, the algorithm identifies requests that are POSTs or GET with parameters (see Line 5- 10). These type of requests are commonly used to upload

---
Algorithm 3 Disconnected node filter

1: **procedure** EXFILTRATIONFILTER(*graph*, τ)
2:     $alerts \leftarrow []$
3:     $exfiltration \leftarrow$ set()
4:     $connections \leftarrow$ map(tuple $\rightarrow$ set())
5:     **for** *node* in *graph* **do**
6:         **if** *node*.method == POST or
7:             *node*.method == GET **then**
8:                 $exfiltration \leftarrow node$
9:         **end if**
10:         $connections$[(method, uri)] $\leftarrow node$
11:     **end for**
12:     **for** *connection, requests* in *connections* **do**
13:         $score \leftarrow$ SIMILAR(*requests*)
14:         **if** $score > \tau$ **then**
15:             $alerts \leftarrow exfiltration \cup requests$
16:         **end if**
17:     **end for**
        return *alerts*
18: **end procedure**
19: **procedure** SIMILAR(*requests*)
20:     $score \leftarrow 0$
21:     **for** $i = 0$ to *requests*.length-1 **do**
22:         $score+ =$dist($requests[i], requests[i+1]$)
23:     **end forreturn** *score*
24: **end procedure**

---

data, and most of the time they are connected into a graph. Second, these requests are checked for similar behavior (see Line 19- 23). Requests are considered similar if they have the same method and URI, and their header values are almost the same. The requests that are considered similar, are then triggered as alert because they represent a repeated attempt of uploading information.

B SIMILARITY FUNCTIONS

$$s_1(F_{a_1}, F_{b_1}) = \begin{cases} 1 & \text{if } F_{a_1} \subseteq F_{b_1}, \\ 0 & \text{otherwise;} \end{cases} \quad (1)$$

$$s_2(F_{a_2}, F_{b_2}) = \begin{cases} 1 & \text{if } F_{a_2} = F_{b_2}, \\ 0.5 & \text{if } F_{a_2} \supset F_{b_2}, \\ 0 & \text{otherwise;} \end{cases} \quad (2)$$

$$s_3(F_{a_3}, F_{b_3}) = \begin{cases} 1 & \text{if } |F_{a_3} - F_{b_3}| \le \varepsilon, \\ 0.5 & \text{if } |F_{a_3} - F_{b_3}| \le 2\varepsilon, \\ 0 & \text{otherwise;} \end{cases} \quad (3)$$

$$s_4(F_{a_4}, F_{b_4}) = \begin{cases} 1 & \text{if } F_{a_4} = F_{b_4}, \\ 0 & \text{otherwise.} \end{cases} \quad (4)$$

$$s_5(F_{a_5}, F_{b_5}) = \begin{cases} 1 & \text{if } F_{a_5} = F_{b_5} \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

C USER DATASET (UD)

Table 4: User Dataset Overview.

User ID	HTTP Requests	Total Hours Traffic	Training Time (hours)	Testing Time (hours)	Trained Fingerprints
1	6739	27	10	17	8
2	4709	48	7	41	3
3	10250	18	7	11	9
4	4480	30	8	22	7
5	20075	35	9	26	93
6	3296	47	9	38	6
7	18525	17	7	10	20
8	41527	125	5	120	8
9	14165	146	10	135	11