Exploiting Temporal Persistence to Detect Covert Botnet Channels

Frederic Giroire[1], Jaideep Chandrashekar[2], Nina Taft[2], Eve Schooler[2], and Dina Papagiannaki[2]

Intel Research[2] and CNRS, France[1]

Abstract. We describe a method to detect botnet command and control traffic and individual end-hosts. We introduce the notion of "destination traffic atoms" which aggregate the destinations and services that are communicated with. We then compute the "persistence", which is a measure of temporal regularity and that we propose in this paper, for individual destination atoms. Very persistent destination atoms are added to a host's whitelist during a training period. Subsequently, we track the persistence of new destination atoms not already whitelisted, to identify suspicious C&C destinations. A particularly novel aspect is that we track persistence at multiple timescales concurrently. Importantly, our method does not require any a-priori information about destinations, ports, or protocols used in the C&C, nor do we require payload inspection. We evaluate our system using extensive user traffic traces collected from an enterprise network, along with collected botnet traces.
We demonstrate that our method correctly identifies a botnet's C&C traffic, even when it is very stealthy. We also show that filtering outgoing traffic with the constructed whitelists dramatically improves the performance of traditional anomaly detectors. Finally, we show that the C&C detection can be achieved with a very low false positive rate.

1 Introduction

A botnet is a collection of compromised end-hosts all under the control of a particular *bot-master* (or *bot-herder*). The recruited end-hosts (also called *drones* or *zombies*) are marshalled and controlled by the bot-herders via a command and control (in short, C&C) traffic channel to carry out a number of malevolent activities. For example, they are used to launch DDoS attacks, send SPAM, harvest personal information from the zombie hosts, stage social engineering attacks, and so on. Botnets are so effective at delivering these "services" that there is an thriving (underground) economy based around buying or renting botnets [1]. Today's commercial malware prevention methods, typically host based HIPS and AV engines, are well suited to identifying and countering previously identified and analyzed threats. However, contemporary botnets are extremely adaptable and able to churn our variants at a very high volume, using polymorphic engines and packing engines, which can easily get around existing defenses (a particular AV vendor reports collecting 3000 distinct malware samples *daily* on average [2]).

In contrast to signature scanning based methods, which target known threats, statistical anomaly detection methods are often employed to detect new threats; these operate by looking for deviations in traffic feature distributions caused by the malware. These methods could possibly detect and flag zombie hosts that have been activated and generating a significant (noticeable) volume of traffic (DDoS attacks, SPAM, click-fraud, etc). However, it may be a considerable period of time between a host joining a botnet to the time that is instructed to carry out a malicious task; often, by then it is too late, as the zombie has served its purpose. A particularly compelling example is the Conficker botnet [3], which has taken over a very large user base since late 2008 but is yet, at the time of this writing, to be used to carry out any particular task. Thus, even as detecting a botnet in the act of performing some detrimental activity should be a goal, it is far more critical to block the initial recruitment vector, or failing that to detect the C&C traffic between the drone and bot-herder, so as to deactivate the channel and render the drone useless. More critically, information gathered about the C&C infrastructure may be used to take down the botnet as a whole.

In this paper, we present and validate a method to detect the C&C communications on an endhost. We were motivated by the observation that a recruited host needs to be in touch with its C&C server to be ready to carry any particular activity botnet. It will reconnect, for each new activity, each time it is repurposed, or resold, and so on. Intuition suggests that such visits will happen with some regularity; indeed without frequent communication to a C&C server, the bot becomes invisible to the bot herder. However, this communication is likely to be very lightweight and spaced out over irregular large time periods. This helps the botnet be stealthy and hence harder to expose. We thus seek to design a detector that monitors a user's outgoing traffic in order to expose malicious destinations that he visits with some temporal regularity, even if infrequently. In order to discern these from normal destinations a user frequents, we build whitelists based on a new kind of IP destination aggregation we call *destination atoms*. A destination atom is an aggregation of destinations that is intended to capture the "service" the user seeks. For example, we view *google.com* as a destination service, because a user's requests will be treated by many different servers with different IP addresses. We build these destination atoms using a series of heuristics. Then, to capture the nebulous idea of "lightweight repetition", we introduce a measure called *persistence*. Our whitelists contain destination atoms that exhibit a given level of persistence. With this whitelist in place, detection proceeds by tracking persistence to contacted (non whitelisted) destinations. When the computed persistence becomes high enough, the destination is flagged as a potential C&C endpoint.

The regularity with which a zombie contacts its bot-herder will differ from bot to bot; moreover, we cannot predict the communication frequency that will be used in tomorrow's botnet. We therefore propose to track persistence over multiple timescales simultaneously so as to expose a wide variety of communication patterns. We develop a simple and practical mechanism to track persistence over many observation windows at the same time.

There are various styles by which botnets can communicate to command control centers, including IRC channels, P2P overlays, centralized and decentralized C&C channels. Our goal here is to try to uncover the C&C activity for the class of bots that employ a high degree of centralization in their infrastructure and where the communication channel lasts for an extended period. We do not target botnets where the communication between zombie and bot-herder is limited to a few connections, or those where the zombie is programmed to use a completely new C&C server at each new attempt.

We validate and assess our scheme using two datasets; one consists of traces collected directly on endhosts, and the second consists of traces of live bot malware. It is important that the whitelist created be stable in that it requires few updates thus avoiding to annoy the user by asking them too often if a given destination/service belongs in their whitelist. Moreover, it is essential that whitelists be small so that they require little storage and can be searched quickly. Using our data traces from a large corpus of enterprise users, we will show that whitelists based on destination atoms, and persistence, exhibit both of these properties. We then overlay the malware traces on top of the user traces, and run our detectors on a replay of the combined traffic. We manually extracted the C&C traffic from the bot malware traces in order to compute false positives and false negatives. We show that our method identifies the C&C traffic in all the bots we tested. We also demonstrate that there is a nice ramification, or additional use, of persistent destination atoms. We can increase the sensitivity of HIDS traffic anomaly detectors, by first filtering the traffic according to the whitelists. This allows a larger fraction of our endhosts to catch the attack traffic, while also speeding up the overall detection time.

2 Related Work

There are three potential avenues with which we could mitigate the botnet problem as described in [4]: preventing the recruitment, detecting the covert C&C channel, and detecting attacks being carried out by the (activated) drones. A number of previous works has addressed the first avenue ([5, 6] among others), and in this paper we chiefly address the second avenue (and the third, albeit indirectly). Our method detects the covert channel end-points by tracking persistence, and we are able to detect attacks by filtering out whitelisted (normally persistent) traffic and subsequently applying well established thresholding methods, borrowing from the domain of statistical anomaly detection.

In [7] the authors devise a method to detect covert channel communications carried over IRC with a scoring metric aimed at differentiating normal IRC channels from those used by botnets based on counts of protocol flags and common IRC commands. Our own work differs in that we do not require protocol payloads, nor are we restricted to IRC activity. Another detection approach, BotHunter [8], chains together various alarms that correspond to different phases of a host being part of a botnet Our own method does not attempt to identify such causality and is also able to detect botnet instances for which other identifying alarms do not exist. Other approaches to detecting botnet traffic involve correlating traffic patterns to a given destination, across a group of users, as is described in [9]. Our own work is complementary, focusing on the individual end-host (and can thus detect single instances of zombies); we can envision combining the approaches together. Botminer [10] attacks the detection problem by first independently clustering presumed malicious traffic and normal traffic and then performing a cross-correlation across these to identify hosts that undertake both kinds of communication. These hosts are likely to be part of an organized botnet. Our own work differs in that it is primarily an end-host based solution, and we do not attempt to correlate activities across hosts. Also, we do not attempt to identify attack traffic in the traffic stream. We focus purely on the nature of communication between the end-hosts and purported C&C destinations, looking for regularity in this communication.

Orthogonal to the problem of detecting the botnets and their activities, and following the increasing sophistication being applied in the design of the more common botnets in operation today, there has been a great deal of interest in characterizing their structure, organization and operation. A description of the inner workings, specifically, the mechanisms used to run SPAM campaigns is described in [11]. The work in [12] examines the workings of *fast-flux* service networks, which are becoming more common today as a way improving the robustness of botnet C&C infrastructure.

The area of traffic anomaly detection is fairly well established, and many detectors have been proposed in the past [13, 14]. Some methods build models of normal behavior for particular protocols and correlate with observed traffic. An indicator of abnormal traffic behavior, often found in scanning behaviors, is a unusual number of connection attempts that fail, as detailed in [13]. A recent interesting work is that of [14] in which the authors try to identify the particular flow that caused the infection by analyzing patterns of communications traffic from a set of hosts simultaneously. All of these approaches are complementary to our work and we allow for any type of traffic feature based anomaly detector to be integrated into the system we describe. Finally, [15] describes a method to build host profiles based on communication patterns of outgoing traffic where the profiles are used to detect the spread of worms. This is different from our own goals in this paper, which is to detect botnet C&C activity, which is a stealthier phenomenon. A more fundamental difference between our approaches is that we employ a notion of persistence, to incorporate temporal information.

To end this section, we strongly believe, given the severity of the problem, that there is no silver bullet solution to tackling the botnet menace; a combination of mechanisms will be needed to effectively mitigate the threat. We view our own work as complementary to existing approaches that are focused on preventing the botnet recruitment or else protecting against the infection vector.

3 Methodology

A common behavior across different bots is that each zombie needs to communicate regularly with a C&C server. In order to keep the C&C traffic under the radar, most bots keep this communication very lightweight, or stealthy. However because the bot will visit its C&C server repeatedly over time, failing which the bot-herder might simply assume the zombie to be inactive, we are motivated to try to expose this low frequency event. To do this, we introduce a notion called *destination atoms* (an aggregation of destinations), and a metric called *persistence* to capture this "lightweight" yet "regular" communication. We design a C&C detection method that is based upon tracking the persistence of destination atoms. In order to differentiate whether a new destination atom exhibiting persistence is malicious or benign, we need to develop whitelists of persistent destinations that the user or his legitimate applications normally visit.

The intuition for our method is as follows: an end-host, on any particular day, may communicate with a large set of destination end-points. However, most of these destinations are transient; they are communicated with a few times and never again. Yet when traffic from the host is tracked over longer periods, the set of destinations visited regularly is a (much) smaller and stable set. Probably, this set consists of sites that the user visits often (such as work related, news and entertainment websites), as well as sites contacted by end-host applications (such as mail servers, update servers, patch servers, RSS feeds, and so on). If the set of destinations with high regularity is not very dynamic, then such a set leads to a whitelist that requires infrequent updating (once learned). We will see in our user study, that indeed such a whitelist is quite stable. This means that should a new destination appear, one that is persistent and malicious, the event stands out (enabling detection). This is precisely what we expect to happen when an end-host is subverted, recruited into a botnet and begins to communicate with its C&C servers.

In order to keep the whitelists compact and more meaningful, we use a set of heuristics to aggregate individual destination endpoints into *destination atoms*, which are logical destinations or services. For example, the particular addresses that respond to `google.com` vary by location and time, but this is irrelevant to the end user who really only cares about the `google` "service". The same is often true for mail servers, print services, and so on. For our purpose, we primarily care about the network *service* being connected to, not so much the actual destination IP address.

Given a destination end-point (dst**IP**, dstPort, proto), we obtain the atom (dst**Service**, dstPort, proto), by extracting the *service* from the IP address using the following heuristics (Table 1 shows a few mappings from endpoints to destination atoms): (i) If the source and destination belong to different domains, the service name is simply the second level domain name of the destination (e.g., cisco.com, yahoo.com). (ii) If the source and destination belong to the same domain, then the service is the third level domain name (e.g., mail.intel.com, print.intel.com). We differentiate these situations because we expect a host to communicate with a larger set of destinations in its own domain, as would be the case in an enterprise network. (iii) When higher level application semantics are available (such as in enterprise IT departments), we can use the following type of heuristic. Consider the FTP service (in PASV mode). The "service" requires two ports on the destination host, one being port 21, and the other an ephemeral port (say k). Thus, both (ftp.service.com,21,tcp) and (ftp.service.com,k,tcp) reflect the *same service* and for the case of FTP. We thus generalize the destination atom as (ftp.service,com, 21:>1024,tcp). Being able to do this for a larger set of protocols requires a detailed understanding of the particular application's semantics, which is beyond our scope here. In this paper, we use a simple heuristic that approximates this behavior: if we observe more than 50 ephemeral ports being connected to at the same service, we expand the destination atom to include all ephemeral ports. The rationale here is that if the service is associated with ephemeral ports, it is likely that we will observe a port number not seen previously at some time and should allow this as part of the same pattern. (iv) Sometimes a single destination host can provide a number of distinct services, and in this case, the destination port is sufficient to disambiguate the services from each other, even though they may have similar "service names", obtained by a (reverse) DNS lookup. (v) Finally, when the addresses cannot be mapped to names, no summarization is possible, and we use the destination IP address as the service name.

Note that aggregating the destination addresses into atoms can help counter the use of fast flux service networks [16] to serve name requests. Since we are constructing the aggregates based on domain names that are looked up, we will be able to group all the fast fluxed destinations into a single atom since they are logically tied together by the common domain name.

Dest.	Dest. Name	Dest. Atom
(143.183.10.12, 80, tcp)	www.inet.intel.com	(inet.intel.com, 80, tcp)
(134.231.12.19, 25, tcp)	smtp-gw.intel.com	(smtp-gw.intel.com, 25, tcp)
(216.239.57.97, 80, tcp)	cw-in-f97.google.com	(google.com, 80,tcp)
(209.85.137.104, 80, tcp)	mg-in-f104.google.com	(google.com, 80,tcp)

Table 1. Example destination atoms contacted by `somehost.intel.com`. Notice that the `intel` hosts, being in the same domain, are mapped onto the third level domain, and the `google` destinations to the second level domain.

We now define our persistence metric, to quantify the "lightweight" yet "regular" communications to destination atoms. We monitor the outgoing traffic from an end-host using a large sliding time window of size W, which is divided into n time-slots, each of length s. W is an *observation window*, and each time-slot s is a *measurement window* (simply, *bin*). Letting s_i denote the i-th slot of size s in W, we have $W \equiv [s_1, s_2, \ldots, s_n]$. The persistence of a destination atom, denoted as $\mathtt{d}$, in the observation window W is defined as:

$$p(\mathtt{d}, W) = \frac{1}{n} \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathbf{1}_{\mathtt{d},\mathrm{s_i}}$$

where $\mathbf{1}_{\mathtt{d},\mathrm{s_i}}$ has a value 1 if the host makes at least one connection to $\mathtt{d}$ in the time-slot s_i, and 0 otherwise. Thus, an atoms persistence is simply the fraction of time slots where at least one connection was observed. Given a threshold p^*, we say that $\mathtt{d}$ is *persistent* if $p(\mathtt{d}, W) > p^*$ (otherwise, it is termed *transient*).

Because botnets differ from one to another to a great extent, we cannot know a priori the frequency (the term is used loosely here) with which a zombie will contact its C&C server(s). Thus, it is of paramount importance to design a method that can track persistence over several observation windows simultaneously. Note that the persistence of an atom depends upon the sizes of the two windows (W, s) over which it is observed and measured; we use the term *timescale* to denote a particular instance of (W, s). In order to capture persistence over many timescales, we select k overlapping timescales $(W^1, s^1) \subset (W^2, s^2) \subset \ldots \subset (W^k, s^k)$, where (W^1, s^1) is the smallest timescale, and (W^k, s^k) is the largest. Here s^j denotes the slot size at time scale j. (We could define s_i^j as the i^{th} slot in an observation window W^j, however we drop the subscript for simplicity when discussing timescales). For each timescale $(W^j, s^j) : 1 \leq j \leq k$, we compute the persistence $p^{(j)}(\mathtt{d})$ as previously defined. Then, a destination atom $\mathtt{d}$ is *persistent* if the threshold p^* is exceeded in *any one* of the timescales, i.e., $\mathtt{d}$ is a *p*ersistent destination atom iff

$$\max_{\mathrm{j}} p^{(j)}(\mathtt{d}) > p^*$$

We have explicitly chosen not to use direct frequency type measurements (e.g., a low pass filter) because our definition is very flexible and does not require one to track specific multiples of one frequency or another. More importantly, we don't expect these low frequency connection events to precisely align at any particular frequency; our definition allows some slack in where exactly the events occur.

When deciding upon the appropriate timescales, particularly the smallest measurement window s^1, we want it capture "sessions" level behavior (multiple requests to a web server in a short interval are likely to be for a single session). Based on a preliminary analysis of user data, we select $s^1 = 1$ hr (87% of connections to the same destination atom are separated by at least an hour). We also set $s^k = 24$ hours because our training dataset is 2 weeks (in reality, one could use larger windows). With these two boundary slot-lengths, we select four additional intermediate values. Thus, the slot lengths corresponding to the timescales we use in this paper are: 1,4,8,12,16,20 and 24 (hours). The observation window length controls how long we should wait before we conclude as to whether something is persistent (or transient). For convenience, we select $n = 10$. Noting that $W = n \times s$, the 7 timescales used in this paper lie between $(W^{min} = 10, s^{min} = 1)$, which is the smallest timescale, and $(W^{max} = 240, s^{max} = 24)$, the largest (all values described in hours). It should be pointed out that additional timescales can be added dynamically based on evidence of some anomaly at a particular timescale.

In the following, we describe the specifics of how our C&C detection proceeds. First, there is a training stage in which the end-host bootstraps itself by learning a whitelist of destination atoms. The training stage should last long enough for the stable behavior of the end-host to be exposed (perhaps a week or two). After the training stage, the detection stage proceeds. In a sense, the training and detection stages proceed identically. In both, persistence of destinations is tracked and alarms raised when this crosses a specified threshold. The fundamental difference is that in the detection stage, an alarm simply results in the atom being incorporated into the whitelist; in the detection stage, the alarm is exposed to the end-user (or communicated to the central IT console in an enterprise) and asked to take punitive action. In case the alarm is benign, and the user (or administrator) can attest to the destination atom, it is added into the whitelist.

C&C Detection Implementation: To simplify the description, we first describe how we do detection with a single timescale. We track the persistence values of destination atoms that are not in the whitelist. Connections to a destination atom, from the end-host, in a window W, is tracked using a bitmap of n bits (one bit for each timeslot in $s \in W$). If a new outgoing connection is observed in slot s_i, then the entry in the bitmap, for the corresponding slot, is set to 1. We create a separate bitmap, all values initialized to 0, per destination atom when it is first encountered observed. This bitmap updating occurs asynchronously as the outgoing connections are observed. A timer fires every s^{min} minutes (this is the time interval corresponding to the smallest timescale), and all the bitmaps are processed. Here, for each bitmap, the persistence is computed taking into account the last n bits and one of three events occurs: (i) it cannot be determined if the persistence is high enough (not enough samples), and the bitmap is maintained; (ii) the newly updated persistence crosses the critical threshold, p^* (raise an alarm, free up bitmap), or (iii) after enough samples, the persistence is below the threshold (the bitmap is freed up).

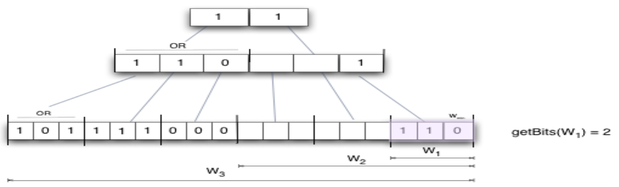

Fig. 1. Bitmaps to track connections at each timescale. Here, we have $n = 3$ and $k = 3$

In order to track persistence at multiple timescales simultaneously, we could use k separate bitmaps per atom. It turns out this is not necessary because we can exploit the structure in our definition of timescales to reduce the overhead. Notice that $s^1 \cdot n = W^1 < W^2 < \ldots < W^k = n \cdot s^k$, that is, s^j is "covered" by a slot in the next higher timescale, as is depicted in Fig. 1. Thus, setting a bit in one of these timescales implies setting a bit in the higher timescale. Thus, rather that maintain separate bitmaps, we can simply construct a single, *long* bitmap that covers all the timescales appropriately. The length of this bitmap is $\frac{W^k}{s^1} = \frac{W^{max}}{s^{min}}$. In our implementation we have $s^1 = 1\ hr$, $n = 10$, and $W^{max} = 240$ hrs, so the bitmap length is exactly 240 bits.

Algorithm 1 computePersistence():

```
1: for all $\mathtt{d} \in \mathtt{DCT}$ do
2:   $p(\mathtt{d}) \leftarrow 0$
3:   for $i = 1$ to $k$ do
4:     $p^{(i)}(\mathtt{d}) \leftarrow \mathsf{getBits}(\mathtt{d}, i) . \frac{|W^{\mathrm{i}}|}{|s^{\mathrm{i}}|}$
5:     $p(\mathtt{d}) = \mathsf{max}(p(\mathtt{d}), p^{(i)}(\mathtt{d}))$
6:     if $p(\mathtt{d}) \geq p^*$ then
7:       RAISEALARM(... suspicious destination d)
8:     end if
9:   end for
10:  $\mathrm{idx} \leftarrow (\mathrm{idx} + 1) \mathrm{mod} \frac{W^{\max}}{s^{\min}}$
11:  if $p(\mathtt{d}) = 0$ then
12:    discard DCT[d]
13:  end if
14: end for
```

High level pseudocode for this entire process is shown in Proc. 1. Here, the set of bitmaps is stored in `DCT`, indexed by individual atoms (line 1 retrieves all the active bitmaps). The loop (lines 2-7) iterates over each timescale, computing persistence in each. There is a separate process that processes each outgoing connection; this checks if the destination is whitelisted, and if not, updates the bit at index `idx` in the bitmap (this index is updated in line 10, each time the procedure is called, i.e., every s^{min}). Finally, if there is no observed activity for the atom in W^{max}, the bitmap is discarded (lines 11-13).

When a C&C alarm is raised, we flag the outgoing connection as suspicious and alert the user who can choose between either adding the destination atom to their whitelist, or blocking the outgoing traffic. We will see in our evaluation that the users are bothered with such decisions infrequently. For enterprise users, such alarms could also be passed to an IT department where they can be correlated with other data.

4 Dataset Description

End Host Traffic Traces: The endhost dataset used in this paper consists of traffic traces collected at over 350 enterprise users' hosts (mostly laptops), over a 5 week period. Users were recruited via intranet mailing lists and newsletters, and prizes were offered as a way to incentivize participation. The results presented in this paper use the traces from 157 of the hosts; these were selected because they provide trace data for a common 4 week period between January and February 2007. Our monitoring tool collected all packets headers, both ingoing and outgoing, from all machine interfaces (wired and wireless). We divide the 4 weeks traffic trace into two halves, a training set and a testing set. The training data is used to build the per-user whitelists and to determine the parameters of our methold (i.e. p^*). The testing data is used to assess the detection performance, i.e., false positives and false negatives.

Botnet Traffic Traces: We collected 55 (known) botnet binaries randomly selected from a larger corpus of malware. Each binary was executed inside a Windows XP SP2 virtual machine and run for as long as a week, together with our trace collection tool. When constructing the clean VM image, we took great pains to turn off all the services that are known to generate traffic (windows auto-update, etc.); we also monitored the VM for a few days on an isolated network to ensure that no IP traffic was being sent out of the system. This gives us a certain level of confidence that all (or nearly all) the traffic collected corresponds to botnet activity. During the collection, the server hosting the VMs was placed behind a NAT device and connected to an active DSL link.

While we expected the trace collection to be a straight-forward exercise, this turned out not to be the case. To begin with, a lot of the binaries simply crashed the VM or else did nothing (no traffic was observed). In other cases, the C&C seemed to have been deactivated, and we only saw failed connections or connections to illegal addresses (indicating that the DNS entries for the C&C servers had been rewired). Only 27 binaries yielded any traffic at all (the collection was aborted if there was no traffic seen even after 48 hours). Of this set, only 12 binaries yielded traffic that lasted more than a day and these usable traces are enumerated in Table 2. Here, the first column is the identifier assigned by ClamAV [17] for the particular botnet binary. Given our limited infrastructure, we were unable to scale to a large number of binaries; however, we endeavored to collect botnet samples with widely different behaviors and temporal characteristics to assure us that our evaluation results hold for a larger population of botnets.

In order to evaluate the effectiveness of our algorithm, we overlay these botnet traces on top of the traffic traces from each user, and then replay this traffic with our detector monitoring for new persistent destination atoms. To assess the true detections, missed detections and false positives, we need to label our traces (thereby establishing ground truth). We thus manually inspected all of our 12 bot traces in order to isolate the C&C traffic from the remainder of the attack traffic. We used BRO [18] to generate connection summaries. Isolating the C&C traffic turned out to be a very tedious process which involved manually breaking down the traffic across ports and destinations of each botnet trace. Due to a lack of space, we cannot enumerate how we did this for the entire set. In summary, we employed a variety of methods including, extracting IRC commands embedded in the streams, looking at the payloads directly of non-IRC communications, and in some cases examining histograms of payload size to extract unusual patterns (i.e. very high chance of small packet sizes consistent across a subset of connections). As an interesting example, consider Trojan.AimBot-5.

First we constructed histograms of traffic to various destinations and on various ports. In this particular case, the communication involved a few destinations. By zooming in on these individual connections and reconstructing the associated TCP streams we obtained a "conversation" between the zombie and the significant destination. We were able to identify IRC protocol commands being tunneled over HTTP to particular destinations. Further analysis revealed that the destination being contacted was hosting a squid proxy, and the IRC commands were being tunneled through. With `STORM`, we were able to pick out the p2p traffic because of a regularity in UDP packets, very different from the other (attack) traffic.

The second column in Table 2 describes the ports and protocols associated with the C&C channel. The third column is a count of the distinct destination atoms seen in the (isolated) C&C traffic. Column 4 shows the range (min to max) of C&C traffic in connections/minute. This confirms the intuition that the covert communications is light in volume and thus volume based detectors would not be suitable to expose this traffic.

ClamAV Signature	**C&C type**	**# of C&C atoms**	**C&C Volume**, min - max
Trojan.Aimbot-25	port 22	1	0-5.7
Trojan.Wootbot-247	IRC port 12347	4	0-6.8
Trojan.Gobot.T	IRC port 66659	1	0.2-2.1
Trojan.Codbot-14	IRC port 6667	2	0-9.2
kTrojan.Aimbot-5	IRC via http proxy	3	0-10
Trojan.IRCBot-776*	HTTP	16	0-1.
Trojan.VB-666*	IRC port 6667	1	0-1.3
Trojan.IRC-Script-50	IRC ports 6662-6669,9999,7000	8	0-2.1 8
Trojan.Spybot-248	port 9305	4	3.8-4.6
Trojan.MyBot-8926	IRC port 7007	1	0-0.1
Trojan.IRC.Zapchast-11	IRC ports 6666, 6667	9	0-1
Trojan.Peed-69 [Storm]	P2P/Overnet	19672	0-30

Table 2. List of sampled Botnet binaries with clear identifiable C&C traffic

5 Evaluation

In this section we results from overlaying the botnet malware traces on top of each user trace, and then emulating our detection algorithm to evaluate the performance of our detector. The two notable results which we discuss further in this section are: (i) our persistence metric based method can indeed pick out the C&C destination atoms in the botnet traces with a very low false positive rate, (ii) the whitelists we construct can significantly boost the detection rates, and improve detection times, of general anomaly detectors.

5.1 System Properties

As mentioned earlier, for our system to work well, the whitelists should have 2 properties. First, they should be stable, that is, require changes very infrequently (so that bot C&C will stand out, and so that user annoyance is kept small). Second, it is nice when they are small as this speeds up the searching activity (filtering outgoing traffic to whitelist contents). Our whitelists will be stable if the rate at which new persistent destination atoms are added to the whitelist is low; and this will be true when much of the user communication is transient. To examine this for our set of users, we compute all the destination atoms for a given user and the persistence value for each atom. The cdf of these across all users is plotted in Figure 2. We see that less than 20% of the destination atoms have a persistence value greater than 0.2; this validates our intuition that transient destinations form the bulk of endpoints a host communicates with. Very few destination atoms exhibit a persistence greater than 60 or 70%. The observation that a user typically has few persistence destination atoms, confirms that building a whitelist for this traffic aggregation is appealing because it is unlikely to be updated often. Recall that our method uses a parameter p^*, that is used both to construct the whitelists in the training stage, and as an alert threshold when monitoring for new C&C destination during detection (testing phase). This plot of user data suggests that selecting a value of p^* anywhere in the range of 50 to 80% will result in a small whitelist, that is likely to require few updates. We select the value of $p^* = 0.6$ because it is in the flat portion of the curve. Note that the number of destination atoms in the whitelist is not very sensitive to the value of p^* (as long as it is above roughly 0.5) suggesting that this parameter is fairly robust in the sense that it need not necessarily be fine tuned. In figure 3, we plot the histogram of whitelist sizes across all the 157 users. The whitelists for almost all the hosts contain 60-140 destination atoms, which is a very manageable size and thus limits any overhead involved in searching whitelists when filtering. So our user data confirms that whitelists constructed of persistent destination atoms will have these 2 attractive properties.

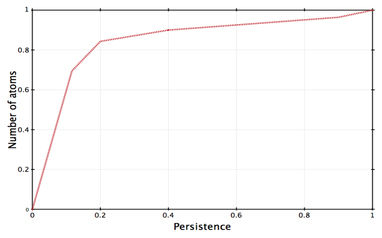

Fig. 2. CDF of $p(d)$ across all the atoms seen in training data

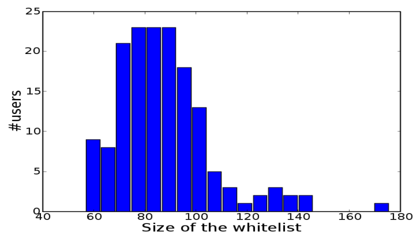

Fig. 3. Distribution of per host whitelist sizes computed using $p^* = 0.6$

5.2 C&C Detection

To assess the ability of our algorithm to identify C&C traffic when it is mixed in with regular user traffic, we overlaid our bot trace data on top each of our

157 user traces. For each user, we replayed the superposed trace and emulated our detector. For each user, we repeated this procedure for each of our 12 bot examples, yielding 1884 tests (157 users times 12 bots). We ran our detector with simultaneous checking for 5 timescales (as indicated in Section 3). The timescales used were with measurement window s taking values $s = (1, 4, 8, 16, 20, 24)$, and the observation window W was always $W = 10s$, i.e. we used (1,10), (4,40), etc. In each of these 1884 instances, our detector was able to correctly identify the C&C traffic. This was validated against our labels (from having isolated the portion of the bot traffic corresponding to the C&C channel). This success illustrates the effectiveness of our persistence metric.

In Table 3 we list various properties of the detected botnets. Column 2 indicates the persistence of a destination atom from a particular bot. Column 3 indicates the timescale that triggered the alert, and the 4th column gives the number of destination atoms that had the properties (persistence and timescale) listed within a row of this table. For example, we see IRCBot-776 listed twice (first two rows of this table) because it used one destination atom that had a persistence of 1 and was detected at a timescale of (10,1), and it had 2 other destination atoms with a persistence of 0.8 that were detected at a timescale of (200,20). This example illustrates that a single bot might used multiple time scales (in terms of how regularly they contact their zombies) for different C&C servers. Looking down column 3, we see that the smallest timescale (10,1) was sufficient to detect at least one of the atoms in all instances except in the case of IRC.Zapchast-11 and Mybot-8926. However, we cannot know ahead of time as to what timescale is appropriate for a particular botnet; thus it is critical to have enough timescales in play to cover a wide range of behaviors. For the STORM bot, we have marked "> 1" in the last column because there a great many of them. The success of our method in uncovering the STORM bot C&C traffic brings up an interesting point: even though our method works best to uncover botnets that tend to have a high degree of centralization, we are able to detect the p2p based infrastructure used by STORM. Thus, our method is likely to be effective at also uncovering non-centralized infrastructures as long as there is a certain repetitiveness in contacting some of the destination atoms involved (out of the thousands, in the case of STORM).

The bot `IRC.Zapchast-11` presents a compelling illustration on how tracking for persistence can be effective even when the connection volume is extremely stealthy. Recall from Table 2 that `IRC.Zapchast-11` generates very little traffic overall - about 1.4 connections per binning interval on average. By all accounts, this is a minuscule amount of additional traffic, that has no chance to stand out in traffic volume against the normal traffic of an end-host, and thus will go undetected by a volume based anomaly detector. However by tracking the persistence of its associated destination atom, we were able to make the anomaly visible. This illustrates the utility of persistence, even in the face of extremely stealthy bots.

Using our two data sets, we can also compute the false positive and detection rate (1 minus false negatives) tradeoff. We computed a traditional ROC curve,

Botnet	**Persistence**	**Timescale**	**# dest. atoms**
IRCBot-776	1.0	(10,1)	1
IRCBot-776	0.8	(200,20)	2
Aimbot-5	1.0	(10,1)	1
Aimbot-5	1.0	(40,4)	1
Aimbot-5	1.0	(160,16)	1
MyBot-8926	0.6	(160,16)	1
IRC.Zapchast-11	1.0	(40,4)	3
Spybot-248	1.0	(10,1)	2
IRC-Script-50	1.0	(10,1)	7
VB-666	0.7	(10,1)	1
Codbot-14	1.0	(10,1)	1
Gobot.T	1.0	(10,1)	1
Wootbot-247	1.0	(10,1)	3
IRC.Zapchast-11	1.0	(10,1)	6
Aimbot-25	1.0	(10,1)	1
Peed-69 [Storm]	1.0	(10,1)	> 1

Table 3. C&C Detection Performance

by sweeping through the full range of values for the cutoff threshold of p^*. The detection rate is computed as the fraction of the tested botnets, across users, for which an alarm is raised. It should be clear that the detection rate is independent of user traffic (the persistence value of an atom does not depend on other atoms). The y-axis denotes an average number of false positives that would be encountered every day. Here, a false positive is simply a destination in the user traffic (assumed clean) which raised an alarm. The values are averaged across all users and over the last two weeks of user traffic data. This RoC curve is shown in figure 4. Earlier we had selected a value for p^* based upon properties of the user generated whitelists. This curve indicates that when the bot traffic is added into the user traffic, then a value of $p^* = 0.6$ is indeed the best choice because it occurs at the knee in this curve. Hence it minimizes the false positives while maximizing the detection rate.

Figure 5 plots the histogram of false positives encountered by all the users, over the entire two week period, as determined by $p^* = 0.6$. A significant fraction of the population see few or no alarms in the two week period. A small handful of users—we speculate these are the very heavy users—see 25-30 alarms over the entire period. To summarize the distribution, we see an average of 5.3 benign destination atoms being flagged as suspicious per user in the 2 week period. That is, the average user will have to dismiss fewer than 1 alarm every other day when this C&C detection system is in place on the users end-host. Since false positives are often associated with user annoyance, our method is able to carry out the detection with an extremely low user annoyance factor. All of our traces being from enterprise users, who are generally well behaved, we cannot really generalize this to other users on the Internet. In fact, it is very possible that applications such as BitTorrent, which connect to a large number of hosts in a short time and this might result in a lot of false positives. Since we do not look inside packet payloads, there is little else that distinguishes a BitTorrent destination from a C&C destination. To get around this, one solution would be possibly whitelist *applications* themselves (which would legitimize all the connections they make).

Since applications like BitTorrent are the exception rather than the norm, we believe that this extended whitelisting will be quite effective.

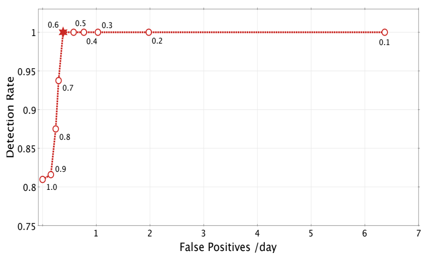

Fig. 4. RoC curve

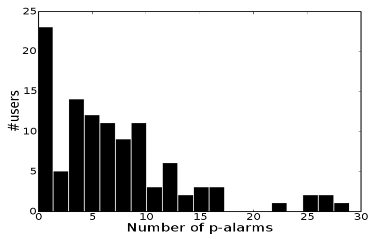

Fig. 5. False positives across users ($p^* = 0.6$)

5.3 Detecting Botnet Attack Traffic

In the previous discussion, we focused on detecting C&C channels. Here, we try to understand how our method can boost the detection rates of more traditional traffic feature (or volume) based anomaly detectors. Note that the whitelists constructed in the training phase can be considered the set of "known good destination" for a particular host. Thus, all traffic going to these destinations must be defacto "anomaly free" and can be *filtered* out of the traffic stream being passed to a conventional anomaly detector.

The traditional anomaly detectors operate by tracking a time series of some significant traffic feature, e.g., number of outgoing connections in an interval, and raising an alarm when this crosses a threshold. The threshold is ideally determined based on the tail of the feature's distribution empirically derived from clean traffic. To distinguish the alarms in question from those triggered by C&C destinations (as discussed previously), we denote them "burst alarms". Thus, the persistence metric results in C&C alarms, and the anomaly detectors generate burst alarms. In the experimental evaluations we describe in the following, the (aggregate, without C&C filtered out) botnet traffic was superimposed on the traffic of each end-host. We point out that the botnet traces are generally shorter than the user traces; to ensure that the trace overlay extends across the user trace, we replicated the botnet trace as often as necessary to fill up the entire testing window of 2 weeks.

There are a number of possible traffic features one can track, and a larger universe of anomaly detectors that can be defined on them. In the current instance, we use a simple connection count detector with a 99.9%-ile threshold. That is, the traffic feature of interest is the number of outgoing connections in 1 minute intervals, and the threshold is computed as the 99.9 percentile value of this distribution empirically computed from the training data. Specifically, we compare the detection results across two traffic streams, one where the outgoing traffic is filtered by the whitelist and the other where it is not. By "filter" we simply mean that all traffic to destinations on the whitelist is ignored and not passed along to the anomaly detector. Note that the same *definitional* threshold is used, i.e., the 99.9%-ile, but the values are different since the time-series are different (one of them has whitelisted destinations filtered out).

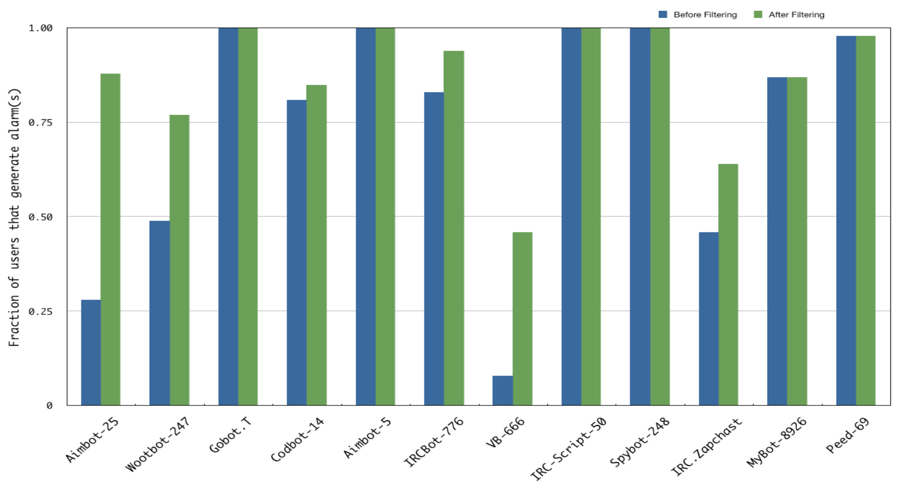

Fig. 6. Improvement in detection rate after filtering

Figure 6 plots the detection rate over the entire user population. The x-axis enumerates the different botnets from Table 2 and the y-axis is the fraction of users that generated a "burst" alarm for the particular botnet. The two bars correspond to the non-filtered and filtered traffic streams, the latter being our enhancement. In the figure, we see a detection rate of 1.0 for some of the botnets, indicating *every* user generated an alarm when fed the tainted trace that included the traffic of (Gobot.T, AimBot-5, SpyBot-50, `storm`/Peed-69). In these cases, the filtering provides no added benefit. This is because the traffic volumes associated with these instances so egregious and beyond the range of "normal" traffic that any reasonable detector would flag this. However, there are lots of other instances in the figure where detection with the filtered traffic is significantly better. For instance, in the case of `VB-666`, which is the most dramatic result, we see a five fold improvement in detection when the traffic is filtered. Another example, with Aimbot-25, only 27% of the users generate an alarm in the general case, but this number grows to 85% when the traffic is filtered— a dramatic improvement. The intuition for why the filtering helps with the detection rate is thus: when the traffic to known good destinations is filtered out and the threshold recomputed, the new threshold tracks the residual traffic better and offers a small "gap" or range that is available for the botnet traffic. That is, as long as the volume of botnet traffic stays inside this gap it will fall under the threshold and be undetected. However this gap is small enough that even a small volume tends to go beyond the usable range. Clearly, the benefit of filtering the traffic is apparent when the botnet traffic volumes are low to moderate. When the volume is large, detection is easily carried out by any reasonably anomaly detector (even without filtering). Thus, filtering traffic through the whitelist helps to uncover "stealthier" traffic that is hidden inside. We carried out the same comparison for other detectors and found the results to be consistent with what we describe here. We omit details from these experiments for a lack of space.

Importantly, notice in the figure that for some of the botnet instances, the detection rate does not reach 100%, even with the filtering. This is possibly because of the variability in traffic across users: presumably, there is a sufficient gap between the traffic and the threshold for some users and the additional botnet traffic is able to squeeze into this gap. However, even when the volume based methods fail to carry out the detection to a complete degree, C&C alarms are generated for *every botnet trace* that we have collected and tested against (as shown in the previous discussion). Thus, even when the attack traffic is small enough to go undetected by the volume detectors, the botnets are still flagged by tracking persistence. This goes to underscore how critical it is to track temporal measures, rather than just volume, when dealing with botnets.

Thus, we demonstrate that by first learning whitelists of "good" destination atoms, and subsequently filtering out the traffic contributions of these destination atoms, we can dramatically improve the detection of botnet associated traffic. We enable more end-hosts to reliably generate alarms for this traffic, and to do so earlier.

6 Conclusions

In this paper, we introduced the notion of "persistence" as a temporal measure of regularity in connection to "destination atoms", which are destination aggregates. We then described a method that builds whitelists of known good destination atoms in order to isolate *persistent* destinations, which are likely C&C channels, in the traffic. The notion of persistence, a key contribution of this work, turns out to be critical in detecting the covert channel communication of botnets. Moreover, being a very coarse measure, persistence does not require any protocol semantics or to look inside payloads to detect the malware.

Using a large corpus of (clean) user traffic as well as a collection of botnet traces, we showed that our method successfully identified C&C destinations in each and every botnet instance experimented with, even the ones that make very few connections. We demonstrated that this detection incurs low overhead and also has a low user annoyance factor. Even though our method is focused on uncovering C&C communication with botnets that have a centralized infrastructure, we are able to uncover even those that are not, as long as there is a certain regularity (even over short time scales) in communicating with the C&C destinations.

In the future, a key task that we would like to undertake is to run our method on a much larger sample of botnet traffic than we were able to collect on our own. Unfortunately, as we learned in the course of this work, such an effort requires a significant amount of resources, and expertise. This goes to show that a much larger community effort is needed to collect, share and annotate traces to support research efforts in designing botnet mitigation solutions.

References

1. de Oliveira, K.C.: Botconomics mastering the underground economy of botnets. FIRST technical colloquium
2. McAfee: McAfee avert labs threat predictions for 2009. `http://www.mcafee.com/us/local_content/reports/2009_threat_predictions_report.pdf`
3. : Conficker working group. `http://confickerworkinggroup.org/`
4. Cooke, E., Jahanian, F., McPherson, D.: The zombie roundup: Understanding, detecting and disrupting botnets. (SRUTI) Workshop (2005)
5. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. IEEE Symposium on Security and Privacy (May 2006)
6. Gao, D., Reiter, M.K., Song, D.: On gray-box program tracking for anomaly detection. In USENIX Security Symposium (August 2004)
7. Binkley, J., Singh, S.: An algorithm for anomaly-based botnet detection. SRUTI Workshop (2006)
8. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. 16th USENIX Security Symposium (Security'07) (2007)
9. Gu, G., Zhang, J., Lee, W.: Botsniffer: Detecting botnet command and control channels in network traffic. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08). (Feb 2008)
10. Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In: 17th USENIX Security Symposium (Security'08). (2008)
11. Kreibich, C., Kanich, C., Levchenko, K., Enright, B., Voelker, G., Paxson, V., Savage, S.: On the Spam Campaign Trail. First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET08) (2008)
12. Holz, T., Gorecki, C., Rieck, K., Freiling, F.: Measuring and Detecting Fast-Flux Service Networks. In: NDSS. (2008)
13. Jung, J., Paxson, V., Berger, A., Balakrishnan, H.: Fast portscan detection using sequential hypothesis testing. Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on (2004) 211–225
14. Sekar, V., Xie, Y., Reiter, M.K., Zhang, H.: Is host-based anomaly detection + temporal correlation = worm causality? Technical Report CMU-CS-07-112, Carnegie Mellon University (March 2007)
15. McDaniel, P.D., Sen, S., Spatscheck, O., van der Merwe, J.E., Aiello, W., Kalmanek, C.R.: Enterprise security: A community of interest based approach. In: NDSS. (2006)
16. Project, G.H.: Know your enemy: Fast-flux service networks. http://www.honeynet.org/papers/ff/
17. : Clamav. `http://www.clamav.net`
18. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks (1999)