LA-UR- 09-06004

Approved for public release; distribution is unlimited.

Title:	AntBot: Anti-Pollution Peer-to-Peer Botnets
Author(s):	Guanhua Yan, Duc T. Ha, Stephan Eidenbenz
Intended for:	The ISOC 17th Annual Network and Distributed System Security Symposium

Form 836 (7/06)

AntBot: Anti-Pollution Peer-to-Peer Botnets

Guanhua Yan† Duc T. Ha‡ Stephan Eidenbenz†

† Information Sciences (CCS-3)
Los Alamos National Laboratory
{ghyan, eidenben}@lanl.gov

‡ Dept of Computer Science and Engineering
University at Buffalo
ducha@buffalo.edu

Abstract

Botnets, which are responsible for many email spamming and DDoS (Distributed Denial of Service) attacks in the current Internet, have emerged as one of most severe cyber-threats in recent years. To evade detection and improve resistance against countermeasures, botnets have evolved from the first generation that relies on IRC chat channels to deliver commands to the current generation that uses highly resilient P2P (Peer-to-Peer) protocols to spread their C&C (Command and Control) information. It is, however, revealed that P2P botnets, although relieved from the single point of failure that IRC botnets suffer, can be easily disrupted using pollution-based mitigation schemes [15].

In this paper, we play the devil's advocate and propose a new type of hypothetical botnets called *AntBot*, which aim to propagate their C&C information to individual bots even though there exists an adversary that persistently pollutes keys used by seized bots to search the command information. The key idea of AntBot is a tree-like structure that bots use to deliver the command so that captured bots reveal only limited information. To evaluate effectiveness of AntBot against pollution-based mitigation in a virtual environment, we develop a distributed P2P botnet simulator. Using extensive experiments, we demonstrate that AntBot operates resiliently against pollution-based mitigation. We further present a few potential defense schemes that could effectively disrupt AntBot operations.

1 Introduction

Botnets, which are networks of compromised machines sharing the same command and control (C&C) infrastructure, have emerged as one of the most severe threats to Internet security in the past few years. To improve resilience to node failures, the new generation of botnets leverage the self-organized structure of P2P networks. Under the umbrella of decentralized P2P network architectures, these botnets can scale up to a large number of nodes but still do not suffer a single point of failure as traditional IRC-based botnets do.

Moreover, these P2P botnets can hide their communications among normal P2P traffic: the botmaster publishes some information (e.g., a command asking every bot to send TCP SYN packets to a target web-server) in a normal P2P network, and each bot, which is also part of the P2P network, regularly retrieves such information with certain features. This technique has been adopted by the recent Storm Worm to hide its communication traffic inside the Overnet network. Under the disguise of normal P2P traffic, botnet C&C communications are much hard to detect at the network layer, given the fact that P2P traffic comprises a large portion of Internet traffic nowadays.

Fortunately, P2P botnets do have their own Achilles' heel. As their C&C communications are based on P2P protocols, they are not immune to those attacks inherent in popular file-sharing P2P networks, where strong authentication is commonly lacked. In [15], Holz et al. explored techniques to mitigate the Storm botnet. They first used honeypots to capture Storm bot executables spread through spamming emails, and then ran these executables in a controlled sandbox. After successfully hooking the controlled bot instances

onto the Storm botnet, they obtained keys that were used by those bots to search the current command issued by the botmaster. It was found that only 32 keys were used every day for the purpose of communications in the Store botnet. By simply overwriting information associated with these 32 keys, the communication of the Storm botnet can be effectively disrupted. This pollution-based technique has been widely used as a sabotage technique to damage the usability of copyrighted contents in file sharing P2P networks [19].

Their results seem promising: P2P botnets, although enjoying the high resilience of decentralized P2P architectures, can be infiltrated and then easily disrupted by pollution-based mitigation schemes. The question, then, is: can P2P botnets be intelligently designed to defeat pollution-based mitigation schemes? In this paper, we play the devil's advocate and design a type of hypothetical P2P botnets called *AntBot* that significantly enhance the resilience of P2P botnets against pollution-based mitigation. In a nutshell, our key contributions made in this paper are summarized as follows: **(1)** We design a distributed protocol for AntBot, which uses a tree-like structure for AntBot to propagate botnet commands in P2P networks. The *key idea* of AntBot is that there are far more low-level bots that are closer to the bottom of the tree than high-level bots so that if a bot is seized by the adversary, it is highly likely that it is a low-level bot and polluting the keys that this bot uses to search or publish the command affects only a small number of bots at lower levels. **(2)** We analytically study three important properties of AntBot: reachability, resilience against pollution, and scalability. Through numerical analysis, we perform sensitivity study of AntBot against its key input parameters. The results provide guidelines for how to configure these parameters in a practical setting. **(3)** We implement AntBot using actual development code of a popular P2P client, aMule, which is based on the KAD protocol, a variant of Kademlia. To evaluate performance of AntBot, we develop a distributed P2P botnet simulator that replaces system calls related to time and socket in the original aMule code with simulated functions. We also implement a crawling-based index pollution scheme in the simulator. **(4)** We perform extensive simulation studies to investigate how resilient AntBot is against pollution-based mitigation schemes. To achieve greater realism in these experiments, we model the churn phenomenon and time zone effects of both regular P2P users and bot activities, using previous measurement results collected from the KAD network and the Storm botnet. The simulation results reveal that AntBot indeed greatly improves the resilience of the P2P botnet against pollution-based mitigation. **(5)** We propose a few potential defense schemes that could effectively disrupt AntBot operations of AntBot and also present challenges that researchers need to address when developing each of these mitigation techniques.

The remainder of this paper is organized as follows. Section 2 presents previous work related to AntBot. In Section 3, we provide details on how to design the protocol for AntBot and use an example to illustrate its operation. In Section 4, we mathematically analyze the performance of AntBot from three different perspectives: reachability, resilience against pollution, and scalability, and also provide numerical results on the effects of different input parameters. Section 5 elaborates on how we develop AntBot using the implementation code of an existing P2P client on a distributed simulation platform. We evaluate the performance of AntBot in 6 and suggest some potential countermeasures in Section 7. Finally, we draw concluding remarks in Section 8.

2 Related Work

As botnets emerge as one of the most severe threats to Internet security, there have been a plethora of works dedicated to botnet research in the past few years. One line of research is focused on analyzing behaviors of real-world bot executables obtained through spamming emails, honeypots, etc. In [2], Barford and Yegneswaran compared four IRC-based botnets, including Agobot, SDBot, SpyBot, and GTBot, from several different perspectives. Behaviors of a few HTTP-based bots, including Rustock, BlackEnergy, and Clickbot.A, have been investigated in [6, 22, 8]. Porras et al. performed static analysis of the Storm worm executable [23], Holz et al. analyzed the propagation mechanism of the Storm worm and its behaviors at both system and network levels [15], and Kanich et al. unraveled a few interesting myths about the Storm

overnet [17]. Our work was motivated by the observation made in [15] that using a strong pollution scheme, it is possible to disrupt operations of real-world P2P botnets like Storm, but from our work we conclude that a P2P botnet, if carefully designed, could still operate resiliently against pollution-based mitigation.

There are also several measurement studies on existing botnets. Rajab et al. used a honeypot to track 192 IRC-based botnets and made some interesting observations on their spreading and growth patterns [1]. The Torpig botnet was hijacked and it was found that this botnet, estimated at consisting of about 182,000 compromised machines, posed severe threats including financial data stealing, DoS attacks, and password leakage [30]. Dagon et al. used a DNS redirection technique to capture botnet traffic and by analyzing such traffic, they found that botnet growth exhibited strong time zone effects [7]. Botnet-based spamming campaigns have been studied by analyzing similarity of email texts [38] and embedded URLs [36]. In this work, when we evaluate resilience of AntBot against pollution-based mitigation, we use results from these previous measurement studies to build realistic models that characterize important aspects of bot activities, such as time zone effects and diurnal patterns.

Many bot and botnet detection techniques have been developed recently. Gu et al. developed detection approaches that hinge on IDS(Intrusion Detection System)-driven dialog correlation [12] and strong spatial-temporal correlation and similarity of bot activities in the same botnet [13][11]. Liu et al. proposed an approach based on virtual machines to detect bot-like activities on individual hosts [20]. Transport-layer communication records have been used to detect botnet behaviors in large Tier-1 ISP networks [18]. Ramachandran et al. developed a botnet detection scheme that relies on passive analysis of DNS-based blackhole list (DNSBL) lookup traffic [25]. Signature-based botnet detection schemes have also been proposed, including Rishi [10] and AutoRE [36]. TAMD is developed by Yen and Reiter that detects stealthy malware, including bots, in an enterprise network by mining communication aggregates in which traffic flows share common characteristics [37]. Our work is orthogonal to these botnet detection efforts and it remains as our future work how to develop effective countermeasures against intelligently designed botnets such as AntBot.

We are not alone in exploring hypothetical botnets that are hard to detect or disrupt. Vogt et al. proposed Super-Botnet, which divides a large botnet into networks of smaller independent botnets to make it more resistant to countermeasures [31]. AntBot, although designed specifically for P2P botnets, can be applied to both large and small botnets to improve their resilience. Overbot, another botnet protocol built on the Kademlia-based P2P networks, aims to hide membership information of the botnet so that a captured bot does not reveal any information about other bots [27]. Chen et al. discussed design of delay-tolerant botnets, which add random delays to command propagation to evade detection [5]. Wang et al. proposed a hybrid P2P botnet with heterogeneous compromised machines (e.g., based on whether they have static IP addresses and whether they are behind firewalls) [32]. Hund et al. provided some guidelines on how to design next-generation botnets that are hard to track and shut down [16]. All these efforts bear different design goals from AntBot, which specifically aims to improve resilience of P2P botnets against pollution-based mitigation. Wang et al. performed a thorough study on P2P botnets in [33], including analyzing effectiveness of index poisoning on mitigating botnet operation. They, however, did not consider how to design botnets intelligently to prevent operation disruption by pollution-based mitigation.

3 Protocol Design of AntBot

P2P Network Model. In this work, we consider the following type of P2P networks. *First*, each peer has a globally unique (or almost unique) identifier. This is applicable to most existing P2P networks because node IDs in them are often randomly generated in a large ID space (e.g., Kademlia, Tapestry, and Pastry), hashed from IP addresses to a large ID space (e.g., Chord), or simply IP addresses of the peers in the network (e.g., Gnutella). *Second*, the P2P network provides two basic primitive operations: $put()$ and $get()$. The

$put()$ operation publishes a data item with a certain key in the network so that other peers can obtain it, and the $get()$ operation instead retrieve a data item with a certain key from the network. In a typical DHT (Distributed Hash Table)-based P2P network, a $put()$ operation stores a data item at a different node whose ID is close to that of the data item; by contrast, in an unstructured P2P network, a $put()$ operation can simply make a local data item accessible to other peers. On the other hand, a $get()$ operation in a DHT-based P2P network searches a data item with a certain ID either iteratively or recursively so that the distance to the target node decreases monotonically. A $get()$ operation in an unstructured P2P network typically involves flooding to get a response, hopefully, from a peer that owns the searched data items. *Third*, there is no strong authentication scheme deployed so that pollution attacks can take place. In such an open network, every peer can publish data items with whatever keys or descriptive strings. This holds for the majority of popular P2P networks these days but not for Freenet, which employs personal namespaces to prevent pollution attacks: storing a data item in a personal namespace requires the private key of its owner, and other peers use the public key of the owner of a namespace to access data items in it. *Fourth*, the P2P network has the churn phenomenon: peers can join or leave the network dynamically.

Algorithm Description. In our design, we assume that the botmaster and all the bots share the same secret key $\mathcal{K}$. We assume that this key is unknown to the adversary[1]. To thwart efforts to obtain key $\mathcal{K}$ by static code analysis, this key is changed regularly by updating the bot executable. The bot executable can also apply sophisticated obfuscation techniques such as polymorphism to complicate static code analysis.

Commands issued by the botmaster are stored as data objects in the P2P network. The keys used to search these data objects are called *command keys*. These command keys are changed regularly every δ time units. For instance, command keys in the Storm botnet change every day. Similar to the Storm botnet, all bots in AntBot use the Network Time Protocol (NTP) to synchronize the time. Without loss of generality, we assume that all bots have the knowledge of a global time t. This global time, for example, can be the Greenwich Mean Time (GMT). Moreover, the global time is discretized into equal periods of length δ time units, which are denoted as $\{\Delta_i\}_{|i=1,2,...}$. The starting time of period Δ_i is denoted as $\tau(\Delta_i)$.

In contrast to the Storm botnet, however, bots in AntBot do not use the same set of keys to search the current command within each period. Let set $B = \{b_k\}_{|k=1,2,...,m}$ denote the entire set of bots in a botnet with m bots, and I_k denote the identifier of bot b_k. For the current time period Δ_i, each bot b_k computes its *signature* $S_k^{(i)}$ as follows:

$$S_k^{(i)} = f(I_k \parallel \tau(\Delta_i)), \tag{1}$$

where $\parallel$ is the concatenation operation and function $f(\cdot)$ is a hashing function that maps the input string into a space of size 2^l. That is to say, the output of function $f(\cdot)$ is an l-bit binary number. For AntBot, it is not required to have a fixed l. Instead, as more bots are recruited into the botnet, the bot executable can be updated to adopt a larger l. Under such circumstance, hash function f should be able to produce digests with a variable length. In this work, we simply use the first l bits from the output of MD5 hash function.

Based on the signature of each bot, we further define its *rank* as follows. First, we define a set of numbers $\{h_j\}_{|0\leq j\leq r_{max}+1}$, where h_j is called a *landmark* and

$$0 = h_0 < h_1 < h_2 < ... < h_{r_{max}-1} < h_{r_{max}} = 2^l. \tag{2}$$

If the signature of a bot falls within $[h_j, h_{j+1})$, its rank is $j+1$. Counterintuitively, we say that rank r_1 is *higher* than rank r_2 if $r_1 < r_2$. All the landmarks are defined in the bot executable.

Let c_i denote the command issued by the botmaster within period Δ_i. To issue the command, the botmaster can remotely login to any compromised machine in the botnet, possibly through multiple step stones to evade detection. He then creates a data object with contents as $E_{\mathcal{K}}[c_i \parallel \tau(\Delta_i)]$, where $E_{\mathcal{K}}[X]$

[1]Throughout this paper, we use *an adversary* to refer to a white-hat security expert attempting to disrupt botnet operation.

denotes encrypting X with key $\mathcal{K}$ using a symmetric encryption algorithm (e.g., DES). We call this data object a *command object*. We further define function g as follows:

$$g(\tau(\Delta_i), d, s) = E_{\mathcal{K}}[\tau(\Delta_i) \parallel d \parallel s], \tag{3}$$

where both d and s are integers, $0 \leq d < 2^l$, and $0 \leq s < s_{max}$. The command object is published every ω time units by the botmaster using the $put()$ operation with command keys in set A, where

$$A = \{g(\tau(\Delta_i), d, s) \mid \forall d : h_0 \leq d < h_1, \forall s : 0 \leq s < s_{max}\}. \tag{4}$$

We say that s is the *slot number* of the command key $g(\tau(\Delta_i), d, s)$. Obviously, $h_1 \cdot s_{max}$ command keys are initially generated by the botmaster. The behavior of the botmaster is illustrated in Algorithm 1.

Algorithm 1 Botmaster's behavior during period Δ_i

```
Login to any bot-controlled machine
Create command c_i for the current period Δ_i
Create the command object as E_K[c_i || τ(Δ_i)]
A ← ∅
for d = h_0 to h_1 − 1 do
    for s = 0 to s_max − 1 do
        Create a command key g(τ(Δ_i), d, s) as in Eq. (3)
        A ← A ∪ g(τ(Δ_i), d, s)
    end for
end for
for every ω time units do
    If it is not in period Δ_i, exit
    Publish the command object with command keys in set A
    Sleep until the next cycle
end for
```

On the other hand, a bot first computes its signature and then its rank r within period Δ_i. The behavior of a bot during period Δ_i is given in Algorithm 2. The bot randomly chooses q distinct numbers from $[h_{r-1}, h_r)$, which are denoted as x_1, x_2, ..., x_q. For each x_j, where $1 \leq j \leq q$, the bot randomly chooses a slot number y_j from $[0, s_{max} - 1]$, and then creates the corresponding command key $g(\tau(\Delta_i), x_j, y_j)$. Note that the landmarks h_1 through $h_{r_{max}-1}$ are chosen in such a way that $h_{r+1} - h_r$, where $0 \leq r \leq r_{max} - 1$, must be no smaller than q. This ensures that each active bot be able to use q command keys to search the command object.

The bot iteratively uses the $get()$ operation to search the command object with command keys in set $B = \{g(\tau(\Delta_i), x_j, y_j)\}_{|1 \leq j \leq q}$. When it finds the data object, it decrypts this object with key $\mathcal{K}$ and checks whether the decrypted time matches the starting time of the current period. If they do not match, the bot keeps waiting for the next searched result; otherwise, it obtains the current command c_i and stops searching. It is noted that time checking in AntBot provides a level of authentication: If the data item is corrupted, say, due to pollution, it is highly likely that the two times do not match and the bot simply ignores the command decrypted from the corrupted data item.

Moreover, an active bot may not be able to find the command object because it has not been published by bots of higher ranks. The bot thus keeps searching the command object with *the same set of command keys* within period Δ_i. The rationale behind such a design is that within period Δ_i, at most q command keys are used by each bot to search the command, thereby limiting the impact of pollution if a bot is seized.

Algorithm 2 Behavior of bot b_k during period Δ_i

Compute $S_k^{(i)}$ as in Eq. (1) and decide its rank as r
Randomly choose q distinct signatures from $[h_{r-1}, h_r)$, denoted as x_1, x_2, ..., x_q
$B \leftarrow \emptyset$
for $j = 1$ to q **do**
    Randomly choose a slot number y_j from 0, 1, ..., $s_{max} - 1$
    $B \leftarrow B \cup g(\tau(\Delta_i), x_j, y_j)$
end for
for every ω time units **do**
    If it is not in period Δ_i, exit
    Search the command object with command keys in B
    SLEEP-MODE:
    Sleep until the next cycle; during this period, if a search result arrives, go to SEARCH-RESULT
end for

{Do the following when a search result arrives}
SEARCH-RESULT:
$V \leftarrow$ data object returned
Use key $\mathcal{K}$ to decrypt V and get command c_i and time t_i
if time t_i matches $\tau(\Delta_i)$ **then**
    Execute command c_i
    Goto COMMAND-FOUND
else
    Goto SLEEP-MODE
end if

{The command is found}
COMMAND-FOUND:
$C \leftarrow \emptyset$
for each d that satisfies the condition in Eq. (6) **do**
    Randomly choose an integer s from 0, 1, ..., $s_{max} - 1$
    $C \leftarrow C \cup g(\tau(\Delta_i), d, s)$
end for
for every ω time units **do**
    If it is not in period Δ_i, exit
    Publish V with command keys in set C
    Sleep until the next cycle
end for

After a bot of rank r successfully derives the current command, it executes this command and publishes it for bots of rank $r+1$. These two things can be done in parallel, especially when executing the command takes a significant amount of time to finish. Note that there are $h_{r+1} - h_r$ unique signatures of rank r. To explain the publishing behavior of a bot, we define $\gamma(r)$, the *branching factor* from rank r, as follows:

$$\gamma(r) = \begin{cases} \frac{h_{r+1}-h_r}{h_r-h_{r-1}} & \text{if } 0 < r < r_{max} \\ 0 & \text{if } r = r_{max} \end{cases} \tag{5}$$

Consider a bot of rank r whose signature is x. If $r < r_{max}$, it publishes the command object with a command key for each signature d of rank $r+1$ that satisfies the following condition:

$$h_r + \gamma(r)(x - h_{r-1}) \leq d < h_r + \gamma(r)(x - h_{r-1} + 1) \tag{6}$$

For each signature d that satisfies the above condition, the bot randomly chooses a slot number s from $[0, s_{max} - 1]$, creates a command key $g(\tau(\Delta_i), d, s)$, and then uses it to publish the command object. It is noted that each bot uses *the same set of command keys* to publish the command object within period Δ_i. This also helps reduce the impact if this bot is seized by the adversary.

Discussion. In practice, the landmarks are set in such a way that branching factors $\gamma(r)$ are greater than 1 except for the lowest level. That is to say, commands are delivered to bots through a tree-like structure, in which commands are relayed by bots at the top to those at the bottom. This is crucial to preventing pollution-based mitigation: because there are more bots with low ranks than those with high ranks, the adversary is more likely to catch a low-rank bot than a high-rank one, thus limiting the impact if he pollutes the keys used by this bot to search or publish the command object. Moreover, each bot uses multiple command keys to search the command object. Although this enables the adversary to pollute more than one command keys if it is caught, the probability that the adversary can catch so many other bots that all command keys used by this bot (if it is not caught) to search the command object are corrupted is also low. Similarly, having multiple slots for each signature reduces the impact of pollution if a bot is caught and all the command keys that it uses to publish the command object are corrupted by the adversary.

Example. We use a simple example, illustrated in Figure 1 and Table 1, to explain the algorithm. In this example, we have: $l = 4$, $q = 2$, $h_1 = 2$, $h_2 = 6$, and $h_3 = 14$. The branching factors from both ranks 1 and 2 can be easily computed as 2; the branching factors from ranks 3 and 4 are 0.25 and 0, respectively. In Steps 1-4 shown in Table 1, the botmaster publishes the command with four command keys, A-D. Bot 1, by randomly choosing a signature from $[0, h_1 - 1]$ and a slot number between 0 and 1, creates command key D and then uses it to get the command (Step 5). Bot 1 further publishes the command with command keys E and F (Steps 6 and 7). Bot 2, on the other hand, generates two command keys L and F and use them to search the command object. Only with command key F can it get the command object (Steps 8 and 9). Bot 2, thereafter, publishes the command object with two other keys, G and H (Steps 10 and 11).

4 Analysis

In this previous section, we have described the algorithm for the botnet operation. We proceed to analyze its performance in this section, particularly from three perspectives: *reachability*, *resilience to pollution*, and *scalability*. Throughout these analysis, we will understand how sensitively AntBot performs under different parameter settings. In our analysis, we consider a botnet with n bots, among which the fraction of active bots within period Δ_i is α_i. Hence, the number of active bots within period Δ_i is $\alpha_i n$.

We also need to know the distribution of the active bots over the 2^l signatures, which is decided by the choice of hash function f. A reasonable assumption might be that an active bot is distributed to each signature with an equal probability. Define vector $\vec{W}$ as follows: $\vec{W} = \langle w_0, w_1, ..., w_{2^l-1} \rangle$, where w_d

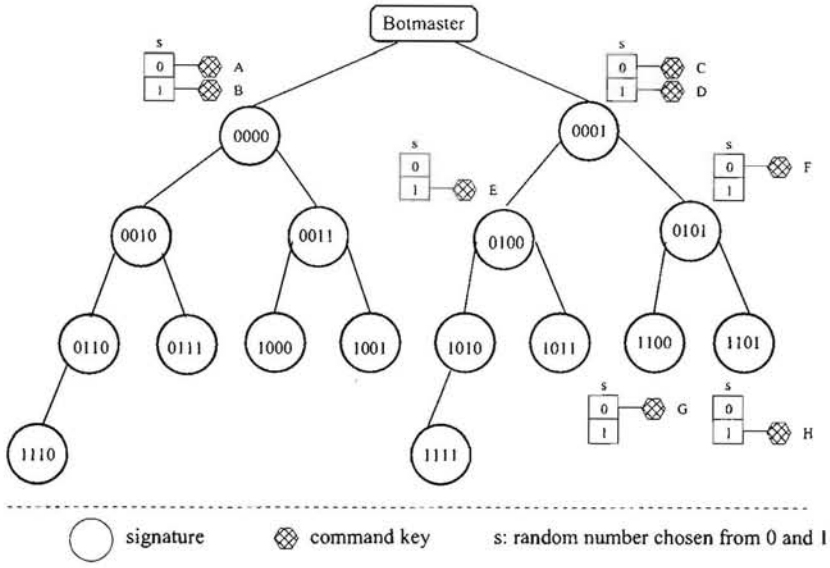

Figure 1: High-level illustration of the algorithm

	Entity	Operation	Key	Signature	s	Results
1	Botmaster	$put()$	A	0000	0	Succeed
2	Botmaster	$put()$	B	0000	1	Succeed
3	Botmaster	$put()$	C	0001	0	Succeed
4	Botmaster	$put()$	D	0001	1	Succeed
5	Bot 1	$get()$	D	0001	1	Succeed
6	Bot 1	$put()$	E	0100	1	Succeed
7	Bot 1	$put()$	F	0101	0	Succeed
8	Bot 2	$get()$	L	0011	0	Fail
9	Bot 2	$get()$	F	0101	0	Succeed
10	Bot 2	$put()$	G	1100	0	Succeed
11	Bot 2	$put()$	H	1101	1	Succeed

Table 1: Steps of spreading a command (the steps are not necessarily sequential)

$(0 \leq d \leq 2^l - 1)$ denotes the number of active bots that are associated with signature d. Given the fact that the number of distributions of i identical objects to j distinct recipients is $\binom{i+j-1}{i}$, the number of combinations for $\vec{W}$ is thus $\binom{\alpha n + 2^l - 1}{\alpha n}$. This number grows at least as fast as $\Theta((\alpha n)^{2^l - 1})$ when $\alpha n \gg 2^l - 1$[2]. With a typical botnet with thousands or tens of thousands of active bots and a reasonable l (e.g., $l = 10$), the number of combinations renders our analysis computationally prohibitive. To simplify our analysis, we assume that each signature corresponds to the same number of active bots[3], which is $\alpha n / 2^l$. Hence, we have $w_0 = w_1 = ... = w_{2^l - 1} = \alpha n / 2^l$. Also, we assume that each active bot of the same rank r publishes the command with the same number of command keys except the case when $r = r_{max} - 1$[4]. That is to say, $(h_{r+1} - h_r) \mod (h_r - h_{r-1})$ equals 0 for all r: $1 \leq r \leq r_{max} - 2$.

Let β denote the reachability of an existing data item, which is defined as the probability that it can be obtained by any peer in the P2P network. It is noted that actual P2P networks are dynamic due to arrival

[2]For $\binom{m}{k}$, it grows as $\Theta(m^k)$ when k is small; when $k = \lfloor m/2 \rfloor$, its growth rate is the fastest, which is $\Theta(2^m / \sqrt{m})$.

[3]Without loss of generality, we ignore the trivial cases in which $\alpha n \mod 2^l \neq 0$.

[4]Here, we cannot assume this holds when $r = r_{max} - 1$ because $h_{r_{max}}$ is fixed at 2^l.

and departure of peers and each bot attempts multiple times to retrieve the command object. Modeling such dynamics, however, is difficult. In our analysis, we ignore these details by simply assuming that β is constant throughout each period Δ_i.

4.1 Reachability

An active bot may not retrieve successfully the command issued by the botmaster due to the following reasons. *First*, even if the data item searched by the bot is available in the P2P network, it may not be reached because of limited flooding in unstructured P2P networks or no paths to it in structured P2P networks. *Second*, when a bot of rank r randomly searches a command key generated with signature d and slot number s ($0 \leq s \leq s_{max} - 1$), it is possible that no bots of rank $r-1$ publish the command with this command key at all. Suppose that bots of rank $r-1$ with signature d' are responsible for using this command key to publish the command. Two cases are possible: active bots with signature d' fail to get the command by themselves, or active bots with signature d' get the command successfully but they do not publish the command object with command keys generated from slot number s. Considering both possibilities, we can establish the following theorem (proof is provided in Appendix A):

Theorem 1. *Suppose that there is a botnet with n bots and the fraction of active ones is α. If the following conditions hold: (1) $\alpha n \mod 2^l = 0$ and the number of active bots associated with each signature is $\alpha n/2^l$, (2) $(h_{r+1} - h_r) \mod (h_r - h_{r-1})$ equals 0 for all r: $1 \leq r \leq r_{max} - 2$, and (3) the reachability of an existing data item is β, then the expected number of active bots that successfully executes the botmaster's command after running the protocol as described is:*

$$n_e = \frac{\alpha n}{2^l} \sum_{r=1}^{r_{max}} (1 - \delta_r)(h_r - h_{r-1}), \tag{7}$$

where:

$$\delta_r = \begin{cases} (1-\beta)^q & if\, r = 1 \\ (1 - \beta(1 - (\delta_{r-1} + \frac{(1-\delta_{r-1})(s_{max}-1)}{s_{max}})\frac{\alpha n}{2^l}))^q & if\, r > 1 \end{cases} \tag{8}$$

4.2 Resilience against Pollution

Suppose that an adversary has captured the bot executable and created m bot instances, each of which runs in a controlled environment. For clarity, we call these bots under control of the adversary *subversive bots*, and as opposed to them are *loyal bots*. The adversary monitors all the command keys that subversive bots use to either search or publish some data objects in the P2P network, and then publishes corrupted information with each of these command keys that have been observed. A loyal bot fails to execute the command if all the command keys it uses to search the command have been corrupted by the adversary.

We assume that there are n loyal bots and the fraction of active ones among them is α. Moreover, the signature of a subversive bot is uniformly distributed over the 2^l ones. Let $\mathcal{C}_p$ and $\mathcal{C}_s$ denote the set of command keys that subversive bots use to publish and search the command data object in the P2P network, respectively. In our analysis, we assume that all command keys in $\mathcal{C}_p \cup \mathcal{C}_s$ are corrupted by the adversary.

Now consider any command key $k_c^{(r)}$ that is searched by loyal bots of rank r. The probability that it is searched by a subversive bot is given as follows:

$$\tilde{p}_s(r) = \frac{h_r - h_{r-1}}{2^l} \cdot \left(1 - \frac{\binom{h_r - h_{r-1} - 1}{q}}{\binom{h_r - h_{r-1}}{q}}\right) \cdot \frac{1}{s_{max}} = \frac{q}{2^l} \cdot \frac{1}{s_{max}} \tag{9}$$

As there are m subversive bots, we thus can derive the probability that command key $k_c^{(r)}$ is searched by *any* of the m subversive bots is:

$$\mathbb{P}\{k_c^{(r)} \in \mathcal{C}_s\} = 1 - (1 - \tilde{p}_s(r))^m. \tag{10}$$

On the other hand, the probability that command key k_c is published by a subversive bot is given by:

$$\tilde{p}_p(r) = \begin{cases} 0 & \text{if } r = 1 \\ \frac{1}{2^l} \cdot \frac{1}{s_{max}} & \text{if } r > 1 \end{cases} \tag{11}$$

Note that command keys of rank 1 are published only by the botmaster and thus cannot be published by subversive bots. Similarly, the probability that command key k_c is corrupted because at least one subversive bot uses it to publish the command data object is given as follows:

$$\mathbb{P}\{k_c^{(r)} \in \mathcal{C}_p\} = 1 - (1 - \tilde{p}_p(r))^m. \tag{12}$$

Let ζ'_r be $\max\{0, 1 - \mathbb{P}\{k_c^{(r)} \in \mathcal{C}_s\} - \mathbb{P}\{k_c^{(r)} \in \mathcal{C}_p\}\}$, where $1 \leq r \leq r_{max}$, and we have the following theorem (proof provided in Appendix B):

Theorem 2. *Suppose that there is a botnet with n loyal bots and the fraction of active ones is α. Also suppose that there are m subversive bots from which command keys that are used to search and publish the command object are corrupted. If the following conditions hold: (1) $\alpha n \mod 2^l = 0$ and the number of active loyal bots associated with each signature is $\alpha n/2^l$, (2) $(h_{r+1} - h_r) \mod (h_r - h_{r-1})$ equals 0 for all r: $1 \leq r \leq r_{max} - 2$, and (3) the reachability of an existing data item is β, then the expected number of active loyal bots that successfully executes the botmaster's command after running the protocol as described is:*

$$n'_e \geq n''_e = \frac{\alpha n}{2^l} \sum_{r=1}^{r_{max}} (1 - \delta''_r)(h_r - h_{r-1}). \tag{13}$$

where δ''_r is computed as follows:

$$\delta''_r = \begin{cases} (1 - \beta(1 - \tilde{p}_s(1))^m)^q & if\, r = 1 \\ (1 - \beta\zeta'_{r-1} \cdot (1 - (\delta''_{r-1} + (1 - \delta''_{r-1}) \cdot \frac{s_{max}-1}{s_{max}})^{\frac{\alpha n}{2^l}}))^q & if\, r > 1 \end{cases} \tag{14}$$

4.3 Scalability

As mentioned earlier, the rationale behind the tree-like command distribution in the proposed scheme is to enhance operational resilience to pollution-based botnet mitigation. One however may argue that a much simpler solution may achieve the same goal: the botmaster publishes the command object with a command key that is uniquely generated for each bot. With this scheme, pollution incurs minimal damage to the botnet operation because bots search for the command with different command keys. Albeit being simple, this scheme obviously has scalability issues, as the botmaster of a big botnet has to publish the command object with a large number of command keys. Moreover, this simple solution also renders traceback easier.

The protocol proposed in Section 3 distributes the task of publishing the command object to the army of bots themselves. Based on the protocol description of AntBot, we can easily establish the following theorem regarding the workload of the botmaster and each bot:

Theorem 3. *If the protocol is executed as described, the botmaster publishes the command with $h_1 \cdot s_{max}$ command keys, and each bot searches for the command with q command keys and publishes the command with at most $\max_{r=1}^{r_{max}-1} \lceil \frac{h_{r+1}-h_r}{h_r-h_{r-1}} \rceil$ command keys.*

4.4 Sensitivity Analysis

We now study the effects of different protocol parameters on botnet behaviors. Let μ denote the number of active loyal bots that fall into each signature. The following scenario will be treated as the baseline case: $s_{max} = 4$, $q = 10$, $h_1 = 64$, $\beta = 0.8$, $\mu = 10$, the branch factor is 4 for all ranks except the lowest one, and τ is 10. To understand the impact of each of these parameters, we vary it among a set of values while keeping the others fixed. Moreover, given a specific combination of parameter settings, we derive both n_e in Eq. (7) when no subversive bots exist and n_e'' in Eq. (13) when m, the number of subversive bots is $2^i \times 40$, where $i = 0, 1, ..., 8$.

Effect of s_{max}. We vary s_{max} between 2 and 20 and the n_e and n_e'' as derived are illustrated in Figure 2. Note that the curve corresponding $m = 0$ actually gives n_e, which is the same in Figures 3-7. We observe that generally speaking, increasing s_{max} helps increase the number of active loyal bots that successfully obtain the command, especially when there are a significant number of subversive bots. This is unsurprising because a larger s_{max} means that there are more command keys used to publish the command, thus reducing the adverse effect of polluting the command keys observed from a fixed number of subversive bots. This can also be observed from both Eq. (9) and (11).

The negative impact of increasing s_{max} is that when it is larger than the number of active loyal bots per signature, some command key slots become empty, thus reducing the probability that the command can be accessed successfully, regardless of whether there are subversive bots or not. Hence, we observe that when $m \leq 640$ in Figure 2, increasing s_{max} beyond 10 actually decreases both n_e and n_e''.

Effect of q. We vary q among 1, 5, 10, 15, 20, 25 and 30, and the n_e and n_e'' as derived are illustrated in Figure 3. Increasing parameter q has two effects. On one hand, a larger q means that an active loyal bot can have more opportunities to obtain the command when it is not accessible via some command keys due to reachability issues inherent in P2P networks or content corruption by the adversary. On the other hand, a larger q means that when the adversary uses a fixed number of subversive bots, he will be able to corrupt more command keys. Hence, we observe mixed effects of increasing parameter q in Figure 3.

Effects of h_1 and the branch factor. Figure 4 shows the effect of varying h_1, the number of signatures of rank 1, among 2^i where $i = 0, 1, ..., 10$, and Figure 5 gives the n_e and n_e'' when we change the branch factor from 2 to 30. The general trend is that increasing either h_1 or the branch factor helps increase the number of active loyal bots that obtain the command successfully. This is because a larger h_1 or branch factor makes the total number of ranks smaller, thus reducing the average number of times needed to forward the command by bots of higher ranks. In Figure 5, when the branch factor is larger than 16, the fraction of active bots that successfully execute the command remains stable, because r_{max} is always 2.

From Theorem 3, we however know that the number of command keys published by the botmaster increases linearly with h_1, and the number of command keys published by each bot is the branch factor (except those bots of the lowest rank). Hence, a larger h_1 leads to heavier workload for the botmaster, and a larger branch factor means that each bot has to publish the command with more command keys.

Effect of β. We also vary reachability parameter β from 0.2 and 1, and the results are illustrated in Figure 6. Unsurprisingly, a higher β always leads to a higher number of active loyal bots that successfully obtain the command.

Effect of parameter μ. Figure 7 presents the effect of varying parameter μ among 1, 10, 20, 30, and 40. The general trend is that a larger μ leads to a higher guaranteed fraction of active loyal bots that successfully obtain the command. Recall that an active loyal bot, after getting the command, publishes it at a random command key slot for each of the children signatures. When μ is larger, the probability that a command key slot at a rank other than 1 is published is higher. The effect of increasing parameter μ, however, becomes less prominent as μ becomes significantly larger than s_{max}, which is 4 for all data points in Figure 7.

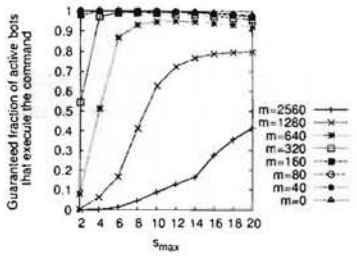

Figure 2: Effect of parameter s_{max}

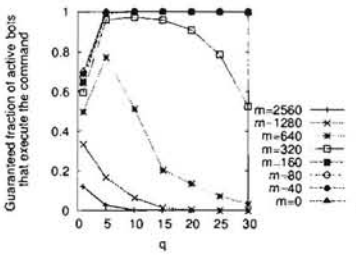

Figure 3: Effect of parameter q

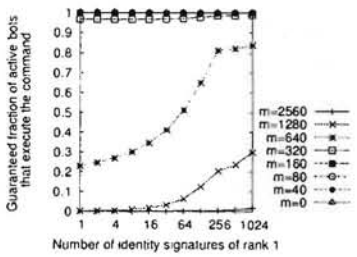

Figure 4: Effect of parameter h_1

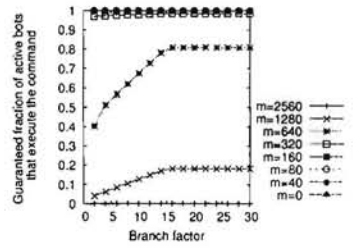

Figure 5: Effect of branch factor

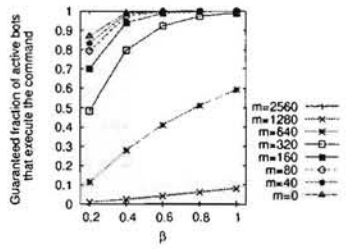

Figure 6: Effect of parameter β

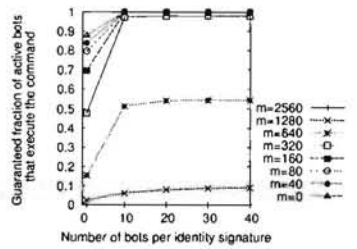

Figure 7: Effect of parameter μ

5 Implementation

In the previous section, we analyzed performance of AntBot under some simplifying assumptions. For instance, we assume that if a command key is polluted by a subversive bot, none of the bots using this key to query the command object are able to receive the command. In reality, this may not be true because the command object stored on some peers may not be polluted by subversive bots. This is confirmed by measurements in [15], which show that even under strong pollution by exhaustive search, a small fraction of Storm bots could still retrieve the command object successfully. To gain a better understanding of AntBot behaviors in a practical setting, we developed a high-fidelity botnet simulator that used actual implementation code of a popular P2P client, *aMule*[5]. aMule implements the KAD protocol, which is a variant of the original Kademlia protocol proposed by Maymounkov and Mazieres [21]. It is noted that the first version of the Storm botnet used the Overnet P2P routing protocol, which is also based on Kademlia. In the following discussion, we first present a brief introduction to Kademlia and KAD; after that, we provide more details on how we implement AntBot with the aMule code base in our distributed simulation testbed.

5.1 Kademlia and KAD

Kademlia is a DHT-based P2P routing protocol, in which each data object or peer is identified by a 160-bit ID. The distinguishing feature of Kademlia is its XOR metric that measures the distance between any two 160-bit identifies x and y: $d(x, y) = x \oplus y$. Data objects are usually stored at those peers whose IDs are close to their owns. Routing in Kademlia is conducted in an iterative manner: when a peer searches for a (node or data object) ID, it queries its neighbors for new peers whose IDs are closer to the target ID; this process repeats until no closer peer IDs can be found.

Although KAD descends from Kademlia, there are some slight distinctions between them. Besides using 128 bits for its node and data object IDs and supporting more diverse messages, KAD uses a two-phase search process. In the first phase, the searching KAD node iteratively queries for peers closer to the target ID but at any time, at most three peers are contacted simultaneously. In this phase, messages of types KADEMLIA_REQUEST and KADEMLIA_RESPONSE are used. After a certain period of time, the search node

[5]The version we used in our study is aMule 2.1.3.

enters the second phase, in which it chooses a few nodes that responded in the first phase and contacts them for the target ID using messages of types KADEMLIA_SEARCH_REQ and KADEMLIA_PUBLISH_REQ. We refer interested readers to [21] and [4] for more details about Kademlia and KAD.

5.2 AntBot Implementation

The basic skeleton of the botmaster's behavior is shown in Algorithm 1[6]. On Line 12, the botmaster needs to publish the command object periodically. In our implementation, we use the *metadata* publishing scheme in KAD to publish the command object. KAD uses a two-level publishing scheme which divides files into two types: metadata and location information. The first level provides references (i.e., location information) to the real data file and the second level uses keywords (i.e., metadata) to fetch location information of real data files. Associated with metadata is a list of tags, such as file names and sizes. Like the Storm botnet, we encode the command object in the filename.

It is, however, noted that the standard KAD protocol allows only one simultaneous metadata publishing at the same time [4]. This is controlled by the KADEMLIATOTALSTOREKEY parameter. Although the bot executable can remove this limitation, our implementation is compliant with the KAD protocol so that it is harder to detect bots by monitoring their behavior. This is done as follows: when publishing metadata, KAD creates a search object with type STOREKEYWORD. As the timeout value of such a search object is 140 seconds, we let the botmaster publish the command object with different command keys every 150 second on Line 13 of Algorithm 1 before going to the sleeping mode. The parameters are properly set so that the number of command keys to publish the command object by the botmaster (i.e., $h_1 - h_0$) does not exceed $\lfloor \omega/150 \rfloor$ (we assume that a time unit is a second here, without loss of generality).

This also applies to the bot behavior shown on Line 35 of Algorithm 2. A bot publishes data object V using different command keys every 150 second. In Algorithm 2, we show that a bot of rank r only publishes $\gamma(r)$ where we recall $\gamma(r)$ denotes the branching factor from rank r (see Eq. (5)). If $\gamma(r)$ is much smaller than $\omega/150$, the bot publishes for only a short period of time every ω time units. From the analytical results in Section 4.1, we know that a bot of a lower rank (i.e., a larger rank number) is less likely to receive the command object because it has to go through more levels of publishing and searching. To improve reachability of the command object to the low-level bots, we slightly modify Algorithm 2 and let each bot publish command objects using command keys in bot sets B and C in the algorithm.

That is to say, a bot also publishes the command object with the command keys that it generated to search the command object. It is easy to see that this does not increase the vulnerability of these command keys: if the bot is seized and thus a subversive bot, the command keys it used to search the command key are known to the adversary anyway. To ensure that a bot has time to publish the command object using command keys in bot sets B and C in Algorithm 2, the bot parameters are set to satisfy: $q + \gamma(r) \leq \omega/150$.

5.3 Pollution

To study how AntBot responds to pollution-based mitigation, we need to implement some pollution mechanisms. Originally, we developed a *passive* pollution scheme: for each subversive bot, the adversary regularly uses the standard KAD protocol to publish junk information (i.e., a random filename in the tag) for each command key that the bot generated to search the command object. This approach, however, does not effectively prevent many loyal bots from obtaining the command object if they persistently search for it.

We thus adopt a more aggressive pollution scheme similar to the one proposed by Holz et al. in [15]. There are two components involved in this approach: *crawlers* and *polluters*. A *crawler* regularly crawls the whole P2P network to obtain a list of active peers. During a crawling cycle, after every three seconds, the crawler sends a route request to 50 new different peers, asking each of them for paths to 16 carefully designed destinations. Once a peer responds to the query by sending a list of peers, the crawler updates its knowledge of current active peers. *Polluters* regularly obtain a list of active peers from the crawlers. To

[6]Here, note that the botmaster's operation can be automatically performed using a script.

prevent overloading a polluter, we let each polluter pollute only a portion of active peers. Every 30 seconds, each polluter selects 100 distinct active peers and publishes on them with junk information using a set of the command keys that the adversary captures using the following two schemes.

Early pollution scheme: When a subversive bot becomes online, the adversary obtains all the command keys it uses to search the command object and sends the command keys immediately to each polluter.

Late pollution scheme: Similar to the early pollution scheme, the adversary monitors the command keys that each subversive bot uses to search the command object. For each subversive bot, the adversary waits for it to get the command object and generate command keys to publish the command object for lower level bots. Once the adversary obtains all command keys that the subversive bot uses to search and publish the command object, it sends them immediately to each polluter.

5.4 Distributed Simulation Testbed

Despite the great realism obtained by using the actual implementation code of a popular P2P client, simulating a large P2P botnet at such a high resolution demands intensive computation. Moreover, the aggressive pollution scheme further significantly increases the number of messages (or traffic) in the network because each crawler exhaustively searches for active peers in the network and each polluter needs to regularly provide junk information for captured command keys. To improve simulation scalability, we develop our simulator on a distributed computing platform. The simulator is a component of MIITS, a local distributed simulation framework for simulating large-scale communication networks [34]. MIITS is built on PRIME SSF, a distributed simulation engine using conservative synchronization techniques [24]. When porting the aMule code into MIITS, we intercept all time-related system calls (e.g., *gettimeofday*) and replace them with simulated time function calls. Similarly, we substitute socket API calls in the original code for network functions developed in MIITS. Moreover, as IP-level routing is not important in this simulation study, we do not model routers on the paths between peers in the P2P network. Previously, we used this simulator to perform a preliminary study of P2P-based botnets and we refer interested readers to [14] for more details.

6 Experimental Evaluation

In this section, we first describe how to model online active durations for both regular peers and bot machines. After providing details of parameter settings in our experiments, we present simulation results that reveal AntBot performance under different scenarios.

6.1 Active Durations of Regular Peers and Bots

To model when a peer or bot joins and departs from the P2P network, we consider both the time zone effect and diurnal patterns observed from previous measurement studies.

Time zone. Time zone effects have been observed from behaviors of normal P2P users [29] and bot activities [7]. To characterize time zone effects in our simulation, we first consider the geographic distribution of the peers in the network. For normal P2P users, we use the following distribution obtained from [29] (Countries are shown as their two-letter country codes defined in ISO 3166-1):

Country	CN	ES	FR	IT	DE	PL	IL	BR	US	TW	KR	AR	PT	GB
Dist.	0.24	0.18	0.12	0.10	0.06	0.04	0.03	0.03	0.03	0.02	0.02	0.02	0.02	0.01

The geographic distribution of bots is generated from the statistics of Storm botnet IP distribution [3]:

Country	US	RU	MX	IN	TR	BR	PL	VN	KR	MA	FR	RO	UA
Dist.	0.22	0.15	0.11	0.09	0.08	0.06	0.05	0.05	0.05	0.04	0.04	0.04	0.03

We assign a country to each normal peer or bot according to the above two tables. If a peer belongs to a country that has multiple time zones (e.g., US), we randomly choose one time zone for it.

Active duration. Once the time zone of each peer has been decided, we further determine its active duration. When a peer is active, it stays in the P2P network and is thus visible to other peers. For the normal peers, we let a small fraction to be always online and active in the P2P network; we call such peers *persistent peers*, as opposed to *transient peers* that join and leave the P2P network regularly. To model behaviors of transient peers, we adopt a model developed in [28] for its active duration. We define the activity cycle of a regular normal peer to be 12:00pm-11:59am. We assume that a regular normal peer is active once in an activity cycle. Its starting time is generated using a Gaussian distribution with mean at 7:00pm and standard deviation as 2 hours. Once the starting time of a normal peer is decided, its active duration is generated using a three-parameter Weibull distribution with the following probability density function:

$$f(x;\lambda,k,\theta) = \begin{cases} \frac{k}{\lambda}(\frac{x-\theta}{\lambda})^{k-1} e^{-(x-\theta/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (15)$$

According to measurement results in [28], we set the parameters as follows: location parameter $\theta = 19.3929$, scale parameter $\lambda = 169.5385$, and shape parameter $k = 0.61511$. With these parameters, the mean active duration is 266.5358 seconds, the same as observed in [28].

Despite observed diurnal patterns of bot activities in the literature [7, 1], no statistical model is ready yet for characterizing active durations of bots. In this study, we use a simple diurnal model mirroring people's normal work hours. We define the activity cycle of a bot machine to be 12:00am-11:59pm. Its starting time is drawn from a Gaussian distribution with its peak at 8:00am and standard deviation as one hour. Similarly, its ending time is drawn from a Gaussian distribution with its peak at 6:00pm and standard deviation also as one hour. Both the starting and ending times of an active duration fall within the current activity cycle.

6.2 Experimental Setup

In our experiments, we study a P2P network with 10,000 peers among which 1000 are bots, either subversive or loyal bots. Among the 9,000 normal peers, there are 1,000 persistent peers that always stay online. As the P2P network takes time to populate the routing table of each peer, we simulate the botnet for three days. The botmaster controls five bot machines, from each of which he sends out a command at the beginning of the third day[7]. For ω in both Algorithms 1 and 2, we let it be 3600 seconds.

In our experiments, we consider three different types of bots. The first type of bots (**baseline-passive**) mirrors behaviors of traditional P2P bots such as Storm bots. When the botmaster uses a machine to release the command, he uses 24 command keys to publish the command object. Each bot randomly chooses three of these command keys to search the command object. A bot does not publish the command object using the command keys it has used to search the command object. The second type of bots (**baseline-active**) differ from baseline-passive bots only for a baseline active bot publishes the command object using the three command keys that it has generated to search the command object. As mentioned earlier, a baseline-active bot does not expose more command keys to the adversary. The third type of bots are **AntBot** as described in Section 5.2. The landmarks defined for AntBot are: 0, 8, 48, and 128. The number of slots for each signature (i.e., s_{max}) is 3 and each bot searches for the command object with 3 command keys. It is noted that the botmaster publishes the command object using the same number of command keys as in the two baseline cases.

We use both the early and late pollution schemes (see Section 5.3) in our experiments. Note that these two pollution schemes are the same for both two baseline cases because a bot does not need to publish the command object for lower level bots. In all our experiments, we assume that the adversary uses two crawlers and five polluters. Also, crawlers and polluters stay online all the time and every half an hour, a crawler sends the peer information it has collected in the past hour to each polluter. We vary the number of

[7]From other experiments, we observe that having more than one machine to send out the command can significantly improve its reachability when there is no pollution.

subversive bots among 0, 10, 20, and 30. Subversive bots, like crawlers and polluters, are always online, and like normal bots, gets activated every 3600 seconds. Each subversive bot sends the revealed command keys to all the polluters when it gets active (early pollution) or gets the command object successfully (late pollution).

For each scenario, we perform five simulation runs with different random number generation seeds. For each simulation run, we use 300 processors on a high-performance cluster and typically a run (if pollution is involved) takes about 13 hours to finish. Crawling-based pollution introduces a significant amount of extra computation time because if no pollution is involved, a simulation run can finish within four hours.

6.3 Experimental Results

In Figure 8, we present the number of bots that successfully receives the command under different simulation scenarios. It is worth noting that in the graph we do not have data points for baseline-passive bots when the number of subversive bots is 10 or 30; hence, the first bar is missing in these two cases.

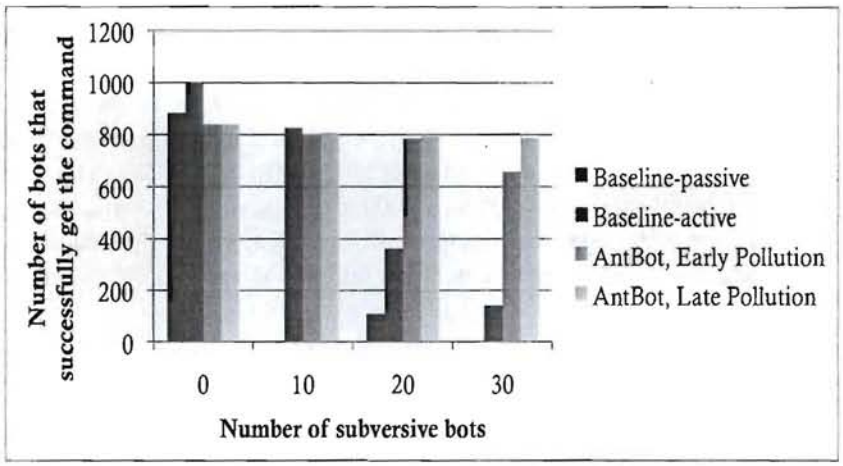

Figure 8: Number of bots that successfully received the command

It is clear from the graph that for the baseline bot, either baseline-passive or baseline-active, when we increase the number of subversive bots in the network, the number of bots that received the command decreases significantly. For instance, when there are 20 subversive bots, only 11% of the baseline-passive bots and 36% of the baseline-active bots can obtain the command successfully; when there are 30 subversive bots, only 14% of the baseline-active bots can get the command. This suggests that pollution-based mitigation indeed adversely affects operation of traditional P2P botnets. On the other hand, having bots publish the command object with command keys that they use to search the command helps deliver the command successfully to individual bots. We can conclude this from the difference in the number of bots receiving the command between baseline-passive and baseline-active bots.

By contrast, AntBot performs much more resiliently against pollution-based mitigation, regardless of the pollution scheme. Even when there exist 30 subversive bots in the network, 66% of bots get the command successfully if the adversary uses the early pollution scheme, and 79% of bots receive the command if the late pollution scheme is applied. In either case, much more bots can get the command than baseline bots. Moreover, although the early pollution scheme pollutes a fewer number of command keys than the late pollution scheme, it obtains the command keys used by subversive bots to search the command immediately after they become active and thus lets polluters to pollute these keys at an earlier time than the late pollution scheme. Therefore, the early pollution scheme seems to be more effective than the late pollution scheme in reducing the number of bots that receive the command, as observed from Figure 8.

The resilience of AntBot comes at a price: under normal circumstances where there are no or few subversive bots, a small fraction of bots cannot obtain the command due to its multi-level command relay mechanism. For instance, even if there is no subversive bot, only 84% of bots can get the command successfully, as opposed to 100% of baseline-active bots and 89% of baseline-passive bots. There are two ways to further improve AntBot. In the *first* approach, the botmaster may want to switch the botnet operation mode to AntBot only when it is found that there exist some subversive bots in the botnet. A bot, when discovering some corrupted messages, reports the situation to the botmaster. This information can be delivered through a data item retrievable by a predefined command key. Obviously, this solution poses another problem: the adversary can pollute this special command key as well. In the *second* approach, both the baseline-active and AntBot command delivery mechanisms are implemented. Normally, each bot uses the baseline-active approach to obtain the command object. Once a bot observes that some command keys have been corrupted by the adversary, it switches to the AntBot mechanism for command propagation. Evaluating performance of such a hybrid mechanism remains as our future work.

7 Countermeasures

From the experimental results shown in Section 6, we know that AntBot functions more effectively than traditional P2P botnets when the adversary pollutes the command keys revealed by subversive bots. In this section, we present three potential countermeasures that can disrupt AntBot operation and also discuss possible challenges when developing these countermeasures.

First, AntBot relies on a secret key shared by both the botmaster and all bots to check whether a data item is the command object or has been corrupted. With ever-improving software reverse engineering techniques, it is possible that the adversary can successfully discover this shared secret key by statically analyzing bot executables. It is, however, another cat-and-mouse game that while the adversary improves his static code analysis skills, the botmaster applies more sophisticated obfuscation techniques such as metamorphism and virtualization [35] to generate bot executables . The botmaster may also apply more advanced PKI (Public Key Infrastructure) techniques (e.g., the Waledac P2P botnet [26]) to prevent botnet disruption due to revelation of shared secrets in bot executables.

Second, as observed from Figure 8, when we increase the number of subversive bots in the network, the fraction of bots that successfully obtains the command still decreases even for AntBot. This is also evident from our analysis in Section 4. Hence, a potential countermeasure against AntBot is to increase the number of subversive bots and thus the number of command keys to pollute. It is easy to mitigate AntBot if each bot, when it runs in a virtual environment, randomly generates its identifier. If this is the case, the adversary can simply run the bot executable in a controlled environment for many times so that a large number of command keys can be revealed. As a response, however, the botmaster may respond by letting each bot executable carry a unique identifier for the bot so that the adversary has to capture many bot executables to derive enough command keys for dismantling AntBot. But this obviously increases the complexity of bot distribution during the propagation process.

Third, given the fact that AntBot is specifically designed against pollution-based mitigation, another possible way of disrupting it is using Sybil-based mitigation. In this approach, the adversary can insert a large number of fake peers (i.e., sybils). These fake peers do not conform to the standard P2P protocol; instead, they attempt to obtain a disproportionately large influence in the P2P network (e.g., they are more likely to be included in the contact list when a peer responds to a KADEMLIA_REQUEST message in KAD). Through these sybils, the adversary can infer bot identities by analyzing which peers search data items with suspicious command keys. The adversary can further provide fake messages to these bots to disrupt botnet operation. Some insights have been provided on how to use sybil-based mitigation to disrupt Storm-like botnets in the literature [9] and it remains as our future work how to use sybil-based mitigation to disrupt AntBot operations.

8 Conclusions

P2P botnets have emerged as a new generation of botnets, whose robustness against one point of failure has significantly improved compared with earlier IRC-based botnets. It is, however, revealed that P2P botnets can be easily disrupted using pollution-based mitigation techniques. In this paper, we play the devil's advocate by exploring possible solutions to improve resilience of P2P botnets against pollution. We propose a new type of hypothetical P2P botnet called AntBot and using extensive simulation, show that AntBot functions well even though the adversary persistently pollutes the command with keys revealed by seized bots. We further present a few potential countermeasures that can effectively disrupt AntBot operations.

References

[1] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of IMC'06*, 2006.

[2] P. Barford and V. Yegneswaran. *Malware Detection*, volume 27 of *Advances in Information Security*, chapter An Inside Look at Botnets. Springer US, 2007.

[3] http://isisblogs.poly.edu/2008/05/19/storm-worm-ip-list-and-country-distribution-statistics.

[4] R. Brunner. A performance evaluation of the kad-protocol. Master's thesis, University of Mannheim, Germany, November 2006.

[5] Z. Chen, C. Chen, and Q. Wang. Delay-tolerant botnets. In *Proceedings of IEEE SecureCPN'09*, 2009.

[6] K. Chiang and L. Lloyd. A case study of the rustock rootkit and spam bot. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, 2007.

[7] D. Dagon, C. C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *Proceedings of NDSS'06.*

[8] N. Daswani and M. Stoppelman. The anatomy of clickbot.a. In *Proceedings of HotBots'07*, 2007.

[9] C.R. Davis, J.M. Fernandez, S. Neville, and J. McHugh. Sybil attacks as a mitigation strategy against the storm botnet. In *Proceedings of Malware'08*, October 2008.

[10] J. Goebel and T. Holz. Rishi: identify bot contaminated hosts by irc nickname evaluation. In *Proceedings of HotBots'07*, 2007.

[11] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of USENIX Security'08*, 2008.

[12] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through ids-driven dialog correlation. In *Proceedings of USENIX Security'07*, 2007.

[13] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of NDSS'08*, 2008.

[14] D. T. Ha, G. Yan, S. Eidenbenz, and H. Q. Ngo. On the effectiveness of structural detection and defense against p2p-based botnets. In *Proceedings of DSN'09*, June 2009.

[15] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *Proceedings of LEET'08*, 2008.

[16] R. Hund, M. Hamann, and T. Holz. Towards next-generation botnets. In *Computer Network Defense, 2008. EC2ND 2008. European Conference on*, Dec. 2008.

[17] C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, and S. Savage. The heisenbot uncertainty problem: challenges in separating bots from chaff. In *Proceedings of LEET'08*, 2008.

[18] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *Proceedings of HotBots'07*, 2007.

[19] J. Liang and R. Kumar. Pollution in p2p file sharing systems. In *Proceedings of IEEE INFOCOM*, 2005.

[20] L. Liu, S. Chen, G. Yan, and Z. Zhang. Bottracer: Execution-based bot-like malware detection. In *Proceedings of ISC'08*, 2008.

[21] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS'01*.

[22] J. Nazario. Blackenergy ddos bot analysis. Technical report, Arbor Networks, October 2007.

[23] P. Porras, H. Saidi, and V. Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. Technical report, Computer Science Laboratory, SRI International, October 2007.

[24] `http://lynx.cis.fiu.edu:8000/twiki/bin/view/Public/PRIMEProject`.

[25] A. Ramachandran, N. Feamster, and D. Dagon. Revealing botnet membership using dnsbl counter-intelligence. In *Proceedings of SRUTI'06*, 2006.

[26] G. Sinclair, C. Nunnery, and B. Kang. Waledac protocol: How and why. In *Proceedings of Malware'09*, Montreal, Canada.

[27] G. Starnberger, C. Kruegel, and E. Kirda. Overbot: a botnet protocol based on kademlia. In *Proceedings of SecureComm'08*, 2008.

[28] M. Steiner, T. En-Najjary, and E. W. Biersack. Analyzing peer behavior in kad. Technical Report EURECOM+2358, Institut Eurecom, France, October 2007.

[29] M. Steiner, T. En-Najjary, and E. W. Biersack. A global view of kad. In *Proceedings of IMC'07*, 2007.

[30] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the ACM CCS'09*, 2009.

[31] R. Vogt, J. Aycock, and M. J. Jacobson. Army of botnets. In *Proceedings of NDSS'07*, 2007.

[32] P. Wang, S. Sparks, and C. C. Zou. An advanced hybrid peer-to-peer botnet. In *Proceedings of HotBots'07*.

[33] P. Wang, L. Wu, B. Aslam, and C. C. Zou. A systematic study on peer-to-peer botnets. In *Proceedings of ICCCN'09*, 2009.

[34] R. Waupotitsch, S. Eidenbenz, J.P. Smith, and L. Kroc. Multi-scale integrated information and telecommunications system (miits): First results from a large-scale end-to-end network simulator. In *Proceedings of WSC'06*.

[35] M. Webster and G. Malcolm. Detection of metamorphic and virtualization-based malware using algebraic specification. *Journal in Computer Virology*, 5(3), August 2009.

[36] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: signatures and characteristics. In *Proceedings of SIGCOMM'08*, 2008.

[37] T.-F. Yen and M. K. Reiter. Traffic aggregation for malware detection. In *Proceedings of DIMVA'08*, 2008.

[38] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, and J. D. Tygar. Characterizing botnets from email spam records. In *Proceedings of LEET'08*, 2008.

Appendix A: Proof of Theorem 1

Let ξ_r denote the probability that the command object is available with a command key that is generated from a signature of rank r and any random slot number between 0 and $s_{max} - 1$. Note that this probability is the same for all such command keys, regardless of the signature and the slot number that are used to generate them, because all signatures of rank r are symmetric and all slot numbers corresponding to the same signature are also symmetric. As the botmaster publishes the command with every signature of rank 1 and every s from 0 to $s_{max} - 1$, we obviously have $\xi_1 = 1$.

Let δ_r denote the probability that an active bot of rank r cannot find any command. It can be simply computed as:

$$\delta_r = (1 - \beta\xi_r)^q. \tag{16}$$

Consider any slot of a signature of rank $r+1$. Potentially, there are $\alpha n/2^l$ active bots of rank r that uses a corresponding command key to publish the command. The probability that each of these bots fails to do so is $\delta_r + (1-\delta_r) \cdot \frac{s_{max}-1}{s_{max}}$. Hence, ξ_{r+1}, the probability that a slot of rank $r+1$ is not empty, is given by:

$$\xi_{r+1} = 1 - (\delta_r + (1-\delta_r) \cdot \frac{s_{max}-1}{s_{max}})^{\frac{\alpha n}{2^l}} \tag{17}$$

With Equations (16) and (17), we can compute δ_r recursively as follows:

$$\delta_r = \begin{cases} (1-\beta)^q & \text{if } r = 1 \\ (1-\beta(1-(\delta_{r-1} + \frac{(1-\delta_{r-1})(s_{max}-1)}{s_{max}})^{\frac{\alpha n}{2^l}}))^q & \text{if } r > 1 \end{cases} \tag{18}$$

Note that there are in total $h_r - h_{r-1}$ signatures with rank r, and there are $\alpha n/2^l$ active bots associated with each of these signatures. As each active bot of rank r successfully executes the command with probability δ_r, we can establish Theorem 1.

Appendix B: Proof of Theorem 2

Let ξ'_r denote the probability that $k_c^{(r)}$ is available and *not* corrupted, and δ'_r denote the probability that an active loyal bot of rank r cannot execute the command successfully. Similar to δ_r in Section 4.1, we have:

$$\delta'_r = (1-\beta\xi'_r)^q. \tag{19}$$

Note that $\tilde{p}_p(1) = 0$ and $\mathbb{P}\{k_c^{(1)} \in C_p\} = 0$. We thus have:

$$\xi'_1 = 1 - \mathbb{P}\{k_c^{(1)} \in C_s\} = (1-\tilde{p}_s(1))^m. \tag{20}$$

Hence, we can calculate δ'_1 as follows:

$$\delta'_1 = (1-\beta(1-\tilde{p}_s(1))^m)^q \tag{21}$$

Different from Equation (17), the calculation of ξ_{r+1} needs to consider the probability of a command key being polluted. We have the following:

$$\begin{aligned} \xi'_{r+1} &= (1-(\delta'_r + (1-\delta'_r) \cdot \frac{s_{max}-1}{s_{max}})^{\frac{\alpha n}{2^l}}) \times (1 - \mathbb{P}\{k_c^{(r)} \in C_s \cup C_p\}) \\ &\geq \zeta'_r \cdot (1-(\delta'_r + (1-\delta'_r) \cdot \frac{s_{max}-1}{s_{max}})^{\frac{\alpha n}{2^l}}) \end{aligned} \tag{22}$$

Hence, for $r \geq 1$, δ'_{r+1} satisfies the following condition based on Equations (19) and (22):

$$\delta'_{r+1} \leq (1-\beta\zeta'_r \cdot (1-(\delta'_r + (1-\delta'_r) \cdot \frac{s_{max}-1}{s_{max}})^{\frac{\alpha n}{2^l}}))^q. \tag{23}$$

Consider δ''_r, which is defined as follows: $\delta''_1 = \delta'_1$, and when $r > 1$,

$$\delta''_r = (1-\beta\zeta'_{r-1} \cdot (1-(\delta''_{r-1} + (1-\delta''_{r-1}) \cdot \frac{s_{max}-1}{s_{max}})^{\frac{\alpha n}{2^l}}))^q. \tag{24}$$

Note that for $r \geq 1$, if $\delta''_r \geq \delta'_r$, then

$$\delta''_{r+1} \geq (1-\beta\zeta'_r \cdot (1-(\delta'_r + (1-\delta'_r) \cdot \frac{s_{max}-1}{s_{max}})^{\frac{\alpha n}{2^l}}))^q \geq \delta'_{r+1}. \tag{25}$$

By induction, we conclude that $\delta''_r \geq \delta'_r$ for all $r : 1 \leq r \leq r_{max}$. Therefore, we thus have:

$$n'_e = \frac{\alpha n}{2^l} \sum_{r=1}^{r_{max}} (1-\delta'_r)(h_r - h_{r-1}) \geq n''_e = \frac{\alpha n}{2^l} \sum_{r=1}^{r_{max}} (1-\delta''_r)(h_r - h_{r-1}). \tag{26}$$

and Theorem 2 follows.