ZEUSMILKER: Circumventing the P2P Zeus Neighbor List Restriction Mechanism

Shankar Karuppayah*¶, Stefanie Roos‡, Christian Rossow§, Max Mühlhäuser*, Mathias Fischer†

* Telecooperation Group
Technische Universität Darmstadt / CASED, Germany
firstname.lastname@cased.de

† Networking and Security Group
International Computer Science Institute, USA
mfischer@icsi.berkeley.edu

§ Cluster of Excellence, MMCI
Saarland University, Germany
crossow@mmci.uni-saarland.de

‡Privacy and Data Security
TU Dresden, Germany
stefanie.roos@tu-dresden.de

¶ National Advanced IPv6 Center,
Universiti Sains Malaysia (USM),
Malaysia

***Abstract*—The emerging trend of highly-resilient Peer-to-Peer (P2P) botnets poses a huge security threat to our modern society. Carefully designed countermeasures as applied in sophisticated P2P botnets such as *P2P Zeus* impede botnet monitoring and successive takedown. These countermeasures reduce the accuracy of the monitored data, such that an exact reconstruction of the botnet's topology is hard to obtain efficiently. However, an accurate topology snapshot, revealing particularly the identities of all bots, is crucial to execute effective botnet takedown operations. With the goal of obtaining the required snapshot in an efficient manner, we provide a detailed description and analysis of the *P2P Zeus* neighbor list restriction mechanism. As our main contribution, we propose ZEUSMILKER, a mechanism for circumventing the existing anti-monitoring countermeasures of *P2P Zeus*. In contrast to existing approaches, our mechanism deterministically reveals the complete neighbor lists of bots and hence can efficiently provide a reliable topology snapshot of *P2P Zeus*. We evaluated ZEUSMILKER on a real-world dataset and found that it outperforms state-of-the-art techniques for botnet monitoring with regard to the number of queries needed to retrieve a bot's complete neighbor list. Furthermore, ZEUSMILKER is provably optimal in retrieving the complete neighbor list, requiring at most $2n$ queries for an n-elemental list. Moreover, we also evaluated how the performance of ZEUSMILKER is impacted by various protocol changes designed to undermine its provable performance bounds.**

I. INTRODUCTION

Cyber-crimes like banking fraud, spam campaigns, and denial-of-service attacks are a profitable business. Most of these attacks originate from botnets, a collection of vulnerable machines infected with malware that are being controlled by a botmaster via a Command-and-Control Server (C2). Traditional botnets utilize a centralized client-server architecture for the communication between the botmaster and its bots. Thus, after such a C2 is taken down, the botmaster cannot communicate with its bots anymore. Recent P2P-based botnets, e.g., *P2P Zeus* [1], *Sality* [2], or *ZeroAccess* [3], adopt a distributed architecture and establish a communication overlay between participating bots. In fact, all attacks against P2P-based botnets require detailed insights into the nature of these botnets, in particular the botnet population and the connectivity graph between the bots [4]. As a consequence, monitoring such botnets is an important task for analysts.

Monitoring P2P botnets requires reverse-engineering of the botnet's malware to at least extract the botnet's communication protocol as well as a seedlist of potential active bots. Afterwards, an analyst can start gathering intelligence about the botnet by either injecting *sensor nodes* or by actively *crawling* it. Sensor nodes [4], [5] can obtain a complete overview of the botnet population, but do not reveal the graph structure of the botnet. However, such connectivity information ("who knows whom") is required to launch successful takedown attempts such as sinkholing [4], [6]. In contrast, crawling relies on graph traversal techniques and thus by design provides the connectivity graph of a botnet. Here, monitoring mechanisms utilize the bots in P2P botnets to reveal information about their neighborhood relationships to increase the view on the connectivity of the network. Initially provided with the contact information of at least one active bot, the crawler iteratively requests neighborhood information from the bots already known to it. In this manner, a snapshot of the botnet's topology can be obtained.

As P2P botnets represent valuable assets to their botmasters, current variants come with additional countermeasures to impede their monitoring. Such measures could be restrictions on the size of exchanged neighbor lists or local reputation mechanisms to detect and blacklist crawlers. One of the most sophisticated P2P botnets known to date that already implements several monitoring countermeasures is *P2P Zeus* [1]. Each bot in *P2P Zeus* maintains a neighbor list that consists of a small subset of other active bots in the network. Bots regularly exchange subsets of these lists on a request basis to maintain and improve the connectivity of the botnet. The exchanged subsets are selected based on the unique keys of the participating bots. Hence, two legitimate bots with two different keys that request a neighbor list from the same other bot, may receive a totally different set of entries. Thus, a botnet crawler has to query each node multiple times using distinct spoofed keys, which decreases the performance of a crawler considerably [7]. Furthermore, *P2P Zeus* utilizes a local reputation mechanism to blacklist IP addresses of bots that request neighbor lists too frequently. In this manner, simple crawlers, e.g., requesting neighbor list using multiple randomly chosen requester keys [4], are prevented from obtaining the

complete neighbor list of a bot. However, exactly this complete neighbor list is vital to launch successful attacks against P2P botnets. Failure to obtain the complete lists, i.e., some nodes left undiscovered, may allow the botmasters to regain control of their botnet through these nodes, hence rendering takedown attempts ineffective.

In this paper, we propose ZEUSMILKER, a novel crawling algorithm that is capable of circumventing the neighbor list restriction mechanism of *P2P Zeus*. ZEUSMILKER exploits the described mechanism for choosing the exchanged subsets of neighbor lists. By requesting a node's neighbor list using strategically chosen spoofed keys as crawler identities, ZEUSMILKER provably retrieves or *milks* (as in milking a cow) the complete neighbor lists of bots at minimal overhead. Although ZEUSMILKER is a specific solution to *P2P Zeus*'s countermeasure, take note that its core idea can be generalized to exploit any domain that utilizes XOR-distance as a metric of '*closeness*', e.g., *Kademlia* [8].

We evaluate the performance of ZEUSMILKER theoretically and via extensive simulations. We prove that ZEUSMILKER obtains the complete neighbor list of size n in at most $2n$ neighbor list requests, thus being optimal with regard to the worst-case complexity. Furthermore, our simulation results indicate that ZEUSMILKER performs significantly better than existing crawling mechanisms.

Finally, to anticipate the next steps of the botmasters, we propose two new countermeasures against crawling. We compare them in a simulation study regarding their ability to impede botnet monitoring with the anti-crawling mechanism built into the *Sality* botnet. Indeed, our simulation results indicate that our countermeasures successfully decrease the crawling performance of ZEUSMILKER and other known crawling mechanisms.

The remainder of this paper is organized as follows: Section II discusses the related work in monitoring P2P-based botnets with emphasis on *P2P Zeus*. Section III introduces our ZEUSMILKER algorithm, its theoretical analysis, and summarizes the findings of an extensive simulation study on crawling *P2P Zeus*. Section IV introduces anti-crawling countermeasures and presents evaluation results on them. Finally, Section V summarizes our contributions and describes the future work.

II. RELATED WORK

P2P botnets are an ongoing research topic. In particular, their architecture and potential takedown strategies have received considerable attention in the research community. We first summarize related work on analyzing existing P2P botnet structures, and afterwards present techniques for botnet monitoring as well as corresponding countermeasures.

Rossow *et al.* provide an overview of eight P2P-based botnet families [4], detailing their architecture. They suggest countermeasures against P2P botnets, such as intelligence gathering and disruption attacks. The *P2P Zeus* botnet, which is the focus of our study, is described in detail in [1]. The authors reverse-engineered the protocol and conclude that *P2P Zeus* is one of the most advanced botnets ever observed.

Monitoring approaches for botnets have been an active topic of research for nearly a decade. Early approaches on crawling the *Storm* botnet [9], [10] focus on finding as many bots as possible, but fail to provide a complete picture of an individual bot's neighborhood relationship. A similar, but more sophisticated approach for enumerating the complete botnet population was proposed by Karuppayah *et al.* in [7] by presenting *LICA*, a generic crawling algorithm that aims to minimize the number of queries needed for enumeration by being as less invasive as possible. Due to its generality, *LICA* does not consider system-specific countermeasures and assume that the complete neighbor list can always be obtained, deferring the question of how to obtain the list to the specific system. Based on this observation, Rossow *et al.* present the only attempt to crawl these bots despite the existing countermeasures [4]. They repeatedly query *P2P Zeus* nodes for their neighbor lists, spoofing different source keys chosen uniformly at random. However, they achieve only limited accuracy and are not able to provable retrieve the complete neighbor list of a bot.

As a reaction to these monitoring activities, researchers proposed botnet designs that prevent disruption and enumeration [11]–[14]. To the best of our knowledge, none of these proposals has been deployed in practice by attackers. In addition, due to lack of prototypical implementations, we could not test if these proposals are actually robust against any kind of enumeration. Instead, we focus on one of the most powerful botnet that *has* been observed in practice, *P2P Zeus*, and design a more efficient algorithm for crawling it.

III. ZEUSMILKER

In this section, we first introduce some background on the *P2P Zeus* botnet. Afterwards, we state and discuss our algorithm for retrieving the entire, i.e., complete, neighbor list from a bot. Finally, we present the evaluation results of our algorithm on a real *P2P Zeus* dataset.

A. System model and notation

We start by giving a general definition of a P2P botnet and introduce the required notation. Then, we explain how this definition is mapped to *P2P Zeus*.

A P2P botnet is characterized by a directed graph $G = (V, E)$, where V is the set of *bots* or *peers*. The set $E \subseteq V \times V$ denotes the connections between bots. In the remainder of this paper we use the terms *bot*, *peer*, and *node* synonymously. Each bot $v \in V$ maintains a neighbor list $\mathrm{NL}_v = \{w \in V | \forall w \in V : (v, w) \in E\}$.

All bots follow a *membership management* protocol that establishes and maintains neighborhood relationships in the botnet to ensure a connected overlay. An important feature of such a protocol is the capability of retrieving information on potential neighbors from other bots to replace unresponsive bots within a bot's own neighbor list. For this purpose, a peer v can request from its neighbor w, a subset of w's neighbor list using the method $\mathrm{requestL}$ that returns a list $\mathrm{L} \subseteq \mathrm{NL}$. The i-th element of a list M is denoted by $M[i]$. In P2P botnet crawling, crawlers exploit this method by iteratively requesting the returned neighbors for their neighbors using the $\mathrm{requestL}$ method until there are no more new nodes returned. From the retrieved neighbor lists, the topology of the network, i.e., the graph $G = (V, E)$, can be reconstructed.

Each bot is assigned a unique *key* in the form of a b-bit string. We use $\mathbf{1}(i)$ to denote a bit string of i $1s$ and analogously $\mathbf{0}(i)$ denotes a string of i $0s$. Furthermore, $|s|$ denotes the length of a string s, and $||$ is the concatenation operator. For two b-bit keys x and y, the function $cp(x, y)$

Algorithm 1: requestL(s)

```
1 for i = 0; i < l && i < |NL|; i++ do
2     L[i] ← NL[i]
3 for i = l; i < |NL|; i++ do
4     for j = 0; j < l; j++ do
5         if XOR(NL[i], s) < XOR(L[j], s) then
6             L[j] ← NL[i]
7             break
8 return L
```

returns their common prefix. An order on the set of b-bit keys is defined by associating the key's bits $b_{b-1} \dots b_0$ with an integer value $\sum_{i=0}^{b-1} 2^{b_i}$. In particular, a key y is defined bigger, smaller or equal than a key x by comparing the integer values. The operators + and − are then defined as the respective operators in $\mathbb{Z}$, the set of all integers. In particular, we call two keys x and y consecutive if $y = x + 1 \mod 2^b$. We use $I(x, y) = \{x + 1, \dots, y - 1\}$ to denote the set of all possible keys '*between*' x and y. Note that the set is empty if $y \leq x$.

B. Neighbor list exchange in P2P Zeus

In *P2P Zeus*, each bot has a unique 160-bit identifier or *key*, which can also be represented as a 40-hexadecimal character string. This key is part of the information stored about any peer's entry in a bot's neighbor list. Although *P2P Zeus* is an unstructured P2P botnet, the entries in a bot's neighbor list are observed to be biased towards the bot's own key. This behavior is influenced by the neighbor list restriction mechanism of *P2P Zeus*. Bots that need information about other bots in the network use the requestL(s) method to ask their neighbors, where s is the key of the requesting bot. However, it is important to note that s can also be generated or spoofed, as long as it is a valid key. On receiving a valid request, a bot returns a subset of its neighbor list of size l, in *P2P Zeus* usually $l = 10$, which are close to key s of the query with regard to the XOR distance. More precisely, as detailed in Algorithm 1, the queried node replies as follows: It first constructs a list L containing up to the first ten elements listed in its neighbor list NL (Line 2). Then, it iterates over all remaining elements in NL (Line 3). The key of each element NL[i] is compared to the current elements of L one-by-one (Line 5). As soon as an element NL[i] with a *smaller* XOR-distance to s than L[j] is found, L[j] is replaced with NL[i] (Line 6). In this fashion, entries or keys with closer XOR-distance are more likely to be returned (Line 5), but only the entry with the *closest* key to s is *guaranteed* to be returned. The second-closest key might not be returned if it is the first element in the initial list L (Line 2). Rather, the queried nodes overwrite the second closest key with the closest key and does not consider it further. So, some closer keys in L might be disregarded despite some non-closest keys being returned. Moreover, the order of the entries stored in a bot's neighbor list is non-deterministic, e.g., they can be sorted by the XOR-distance of the neighbor to the bot or by the timestamp an entry was last updated.

Based on the above observation, we present an algorithm for strategically spoofing keys during crawling, to guarantee the retrieval of the complete neighbor list of a given node.

C. ZeusMilker *algorithm design*

Our goal is to retrieve the complete neighbor list NL of a node using the method requestL(s) for various spoofed keys s. Algorithm 2 achieves our goal by subsequently discovering pairs of keys (x, y) such that the neighbor list NL is guaranteed not to contain any keys in $I(x, y)$ and thus NL $\cap\, I(x, y) = \varnothing$. The algorithm terminates if no set $I(x, y)$ can contain additional and yet unknown keys, guaranteeing that the list of returned keys L is identical with NL. We assume that the neighbor list does not change while crawling it. This assumption is valid because the neighbor list of stable *P2P Zeus* bots typically changes only every 30 minutes during the periodic membership maintenance cycle. We further assume that we are not affected by any rate limiting countermeasures such as that in *P2P Zeus*, as they can be easily circumvented using proxies, i.e., milking the bots in a distributed manner using crawlers with unique IP addresses.

Before discussing Algorithm 2 in detail, we shortly explain how spoofing with two consecutive keys $s_1, s_2 \in I(x, y)$ results in a set $I(x, y)$, such that all keys in $I(x, y)$ are *not* contained in NL. Consider the left-hand side of Figure 1: Here, all possible b-bit keys are represented in the form of a ring. Note that all the keys in the right half of the ring are closer to $\mathbf{0}(b)$ than $\mathbf{1}(b)$ with regard to the XOR distance, whereas all keys on the left half are closer to $\mathbf{1}(b)$. Similarly, when considering only the keys on the right half, the keys in the upper right quarter are closer to $00\|\mathbf{1}(b-2)$ than to $01\|\mathbf{0}(b-2)$, whereas the keys in the lower right quarter are closer to $01\|\mathbf{0}(b-2)$. In this manner, one can successively divide the keys into sets according to which of the two keys they are closer to. We leverage this division to identify the set of keys not contained in the neighbor list NL and the set of keys possibly contained in NL as follows. Let

$$s_1 = c\|0\|\mathbf{1}(i), \quad s_2 = s_1 + 1 = c\|1\|\mathbf{0}(i) \tag{1}$$

for some common prefix c and $i \geq 0$, i.e., s_1 is a key ending with a string of 1s, and s_2 is the next higher key, thus ending with a string of 0s. First note that for any keys id_1 and id_2, XOR(id_1, id_2) starts with a string of 0s of the length of their common prefix. So, if id_1 shares a longer common prefix with id_2 than with a key id_3, id_1 is closer to id_2 than to id_3 with regard to the XOR distance. Now, assume that we know that NL contains keys k_1 and k_2 starting with $c\|0$ and $c\|1$, respectively. As a consequence, x and y, the closest keys in NL to s_1 and s_2 with respect to the XOR distance, have to start with $c\|0$ or $c\|1$, respectively. So, requestL(s_2) returns a list containing a key $y = c\|1\|r_y = s_2 + r_y$ for some i-bit string r_y. Similarly, requestL(s_1) returns a list containing a key $x = c\|0\|r_x = s_1 - \mathbf{1}(i) + r_x$ for some r_x. We now show that indeed x and y are such that NL$\cap I(x, y) = \varnothing$. By the definition of $I(x, y)$, $I(x, y) = I(x, s_2) \cup I(s_1, y)$. The claim that NL $\cap\, I(x, y) = \varnothing$ follows from showing that all $z_x \in I(x, s_2)$ and $z_y \in I(s_1, y)$ have a lower XOR distance to s_1 or s_2 than x or y, respectively, and hence cannot be contained in NL. Note that all $z \in I(x, y)$ share the prefix c. Consider $z_y = c\|1\|q \in I(s_1, y)$ for an i-bit string q, so that XOR$(z, s_2) = q = z - s_2$. As a consequence, we have XOR$(z_y, s_2) <$ XOR(y, s_2) for all keys $z_y \in I(s_1, y)$, so that $z_y \notin$ NL if y is the closest key to s_2 in NL. Similarly, for any $z_x = c\|0\|q \in I(x, s_2)$, $r_x < q \leq \mathbf{1}(i)$, so that XOR$(z_x, s_1) = \mathbf{1}(i) - q$ and hence XOR$(z_x, s_1) <$ XOR(x, s_1). Hence, $z_x \notin$ NL if x is the

Algorithm 2: ZeusMilker()

```
   // Initialization
1  L ← ∅                                  // Crawled keys
   // Get smallest key
2  M ← requestL(0(b))
3  L ← L ∪ M
4  k_first ← getClosestKey(M, 0(b))
   // Get largest key
5  M ← requestL(1(b))
6  L ← L ∪ M
7  k_last ← getClosestKey(M, 1(b))
8  if k_first ≠ k_last && k_first ≠ k_last − 1 then
9      R ← {(k_first, k_last)}   // Undiscovered sets
   // While not fully discovered
10 while not R = ∅ do
       // Get keys for spoofing
11     (k_1, k_2) ← R.pop()
12     c ← getCommonPrefix(k_1, k_2)
13     s_1 ← c||0||1(b − length(c) − 1)
14     s_2 ← c||1||0(b − length(c) − 1)
       // Execute queries and add new sets
15     if k_1 < s_1 then
16         M ← requestL(s_1)      // query with s_1
17         L ← L ∪ M
18         x ← getClosestKey(M, s_1)
19         if x ≠ k_1 then
20             R ← R ∪ {(k_1, x)}
21     if k_2 > s_2 then
22         M ← requestL(s_2)      // query with s_2
23         L ← L ∪ M
24         y ← getClosestKey(M, s_2)
25         if y ≠ k_2 then
26             R ← R ∪ {(y, k_2)}
27 return L
```

closest returned key to s_1. In summary, all keys in $I(x, y)$ are not contained in NL, and we have thus found a method to identify sets of keys that are guaranteed not to be contained in NL. However, without further queries it is not possible to say which keys in $I(\mathrm{k}_1, x)$ and $I(y, \mathrm{k}_2)$ are contained in NL.

Example III.1. As an example, consider the neighbor list NL_{ex} = $\{00000, 00100, 01010, 01100, 10010, 11000\}$ and assume for simplicity that each query via requestL() only returns l = 1 key. Assume we have already discovered k_1 = 00000 and k_2 = 01100 with common prefix c = 0 and now query with s_1 = 0||0||111 = 00111 and s_2 = 01000. requestL(s_1) is guaranteed to return x = 00100 and requestL(s_2) returns y = 01010. However, the reply does not tell us if any keys in $I(\mathrm{k}_1, x)$ = $\{00001, 00010, 00011\}$ or $I(y, \mathrm{k}_2)$ = $\{01011\}$ are contained in NL_{ex}.

Algorithm 2 now subsequently identifies sets of keys which cannot be contained in NL, while at the same time finding new keys k_1 and k_2 that are used for determining the keys s_1 and s_2. Initially, the list of discovered keys L is empty (Line 1). Then s_1 = $\mathbf{0}(b)$ and s_2 = $\mathbf{1}(b)$ are used as keys for the first two queries with the returned list requestL(s_1) and requestL(s_2) added to the set of discovered keys (Lines 2 -7). In particular, requestL(s_1) has to contain the smallest key $\mathrm{k}_{\mathrm{first}}$ and largest $\mathrm{k}_{\mathrm{last}}$ in NL, i.e., the closest keys to $\mathbf{0}(b)$ and $\mathbf{1}(b)$. Hence, the set $I(\mathrm{k}_{\mathrm{last}}, \mathrm{k}_{\mathrm{first}})$ is the first detected set of keys that are not contained in NL. However, $I(\mathrm{k}_{\mathrm{first}}, \mathrm{k}_{\mathrm{last}})$ potentially contains undiscovered keys, given that it is non-empty, i.e., the two keys are not equal or consecutive. So, the pair $(\mathrm{k}_{\mathrm{first}}, \mathrm{k}_{\mathrm{last}})$ is the first element in the set R (Line 9), which contains pairs $(\mathrm{k}_1, \mathrm{k}_2)$ whose common prefix defines the spoofed keys in future iterations. In each iteration of the while loop (Lines 10 - 26), such a pair $(\mathrm{k}_1, \mathrm{k}_2)$ is considered. The common prefix c of k_1 and k_2, determines the two spoofed keys s_1 and s_2, such that $s_1 = c||0||\mathbf{1}(b - \mathrm{length}(c) - 1)$, which consists of the common prefix c, 0, and a string of 1s achieving a total length of b, is the largest key closer to k_1 than to k_2 (in terms of the XOR-distance). Analogously, $s_2 = c||1||\mathbf{0}(b - \mathrm{length}(c) - 1) = s_1 + 1$ is the smallest key closer to k_2 than k_1 (Lines 12-14). If s_1 is not bigger than k_1, $I(\mathrm{k}_1, s_1)$ is empty, so it is not necessary to query with s_1. Analogously, if s_2 is not smaller than k_2, $I(s_2, \mathrm{k}_2)$ is empty. If s_1 is bigger than k_1, the method call requestL(s_1) is executed, the returned list M added to L, and the key x is chosen as the closest key to s_1 in M (Lines 16-18). Similar, if s_2 is smaller than k_2, y is chosen as the closest key to s_2 in the set returned by requestL(s_2) (Lines 22-24). As discussed above, keys in $I(x, y)$ are guaranteed to be not contained in NL, hence only the sets $I(\mathrm{k}_1, x)$ and $I(y, \mathrm{k}_2)$ can possibly contain undiscovered keys, if they are non-empty. Hence, the pairs (k_1, x) and (y, k_2) are added to R (Line 20 and 26, respectively).

Example III.2. We use the exemplary neighbor list NL_{ex} = $\{00000, 00100, 01010, 01100, 10010, 11000\}$ from Example III.1, which is sorted for simplicity and indexed by id_j = $\mathrm{NL}_{ex}[j]$, for j = 0 . . . 5. The ring on the left of Figure 1 depicts how these keys map onto the whole key space. For simplicity, we again assume that only l = 1 keys are returned per query. However, for larger $l < |\mathrm{NL}_{ex}|$, the same number of steps are required to *guarantee* that all keys in NL_{ex} are returned, though individual keys might be discovered much earlier. Initially, two queries are conducted, one with key 11111 (Line 5, Algorithm 2) and one with key 00000 (Line 2, Algorithm 2), which will return two entries from NL, namely $\mathrm{k}_{\mathrm{first}}$ = id_0 = 00000 (Line 4, Algorithm 2) and $\mathrm{k}_{\mathrm{last}}$ = id_5 = 11000 (Line 7, Algorithm 2), respectively. Hence, we know that there are no keys in $I(i_5, i_0)$. Then as described in the following and as can be seen on the right of Figure 1, five iterations of the loop are executed as follows:
1) The pair of keys k_1 = id_0 = 00000, and k_2 = id_5 = 11000 is retrieved from R. They do not share a common prefix, so we spoof with s_1 = 01111 and s_2 = 10000, and discover x = id_3 = 01100 and y = id_4 = 10010. The pairs (id_0, id_3) and (id_4, id_5) are added to the set R. After this step, we can guarantee that NL_{ex} does not contain keys in $I(id_3, id_4)$, since they would have been returned when spoofing IDs s_1 or s_2.
2) The pair (id_4, id_5) = $(10010, 11000)$ is retrieved, sharing common prefix 1. The spoofed keys are thus s_1 = 10111 and s_2 = 11000. Because s_2 is identical to id_5 and hence there are no keys in $I(s_2, id_5)$, it is not necessary to spoof with s_2. Spoofing with s_1 does not result in any closer key to s_1 than id_4. No new pairs are added to R, and it is guaranteed that NL_{ex} does not contain keys in $I(id_4, id_5)$.
3) The pair (id_0, id_3) = $(00000, 01100)$ is processed. Spoofing

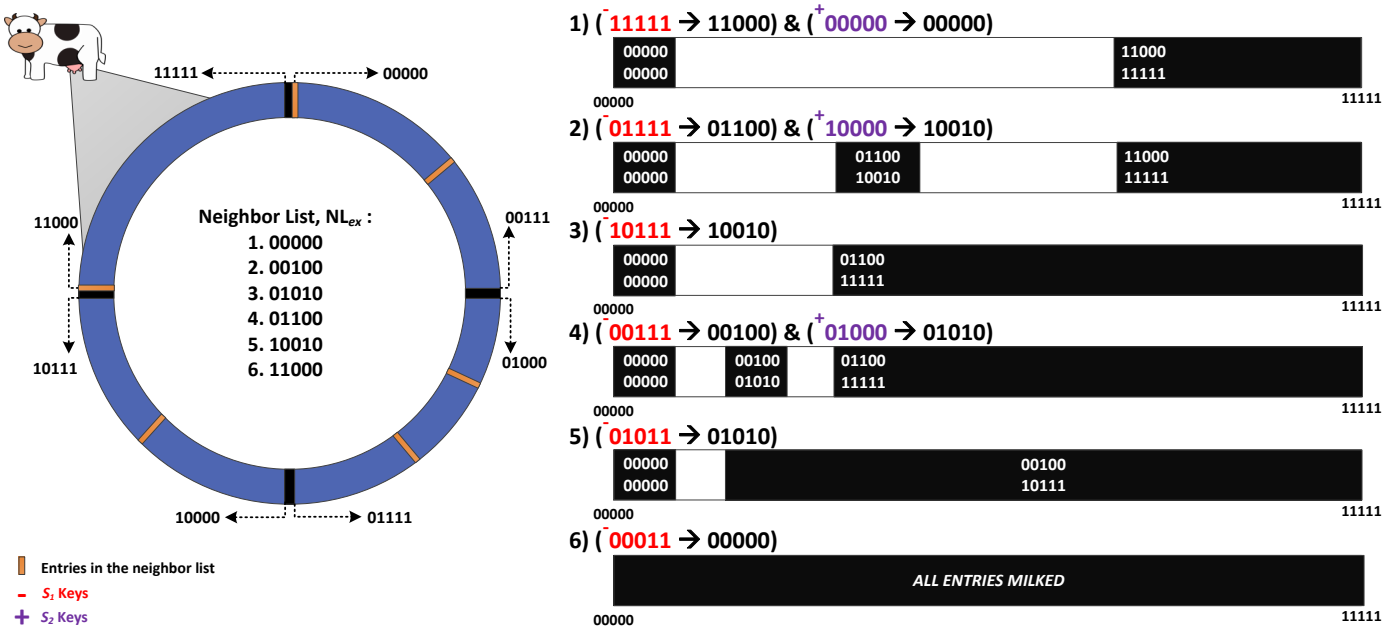


Fig. 1. Visual representation of the key space (Example III.2). The '+' keys discover the next bigger key, whereas the '−' keys reveal the next smaller key.

with $s_1 = 00111$ and $s_2 = 01000$ leads to the discovery of $id_1 = 00100$ and $id_2 = 01010$. Therefore, the pairs (id_0, id_1) and (id_2, id_3) are added to R. As a consequence, we know that NL_{ex} does not contain keys in $I(id_1, id_2)$.
4) The pair $(id_2, id_3) = (01010, 01100)$ is retrieved, but spoofing with $s_1 = 01011$ (spoofing $s_2 = 01100$ not required) reveals that NL_{ex} does not contain keys in $I(id_2, id_3)$.
5) The pair $(id_0, id_1) = (00000, 00100)$ is retrieved, but spoofing with $s_1 = 00011$ (spoofing $s_2 = 00100$ not required) reveals that NL_{ex} does not contain keys in $I(id_0, id_1)$.

The example indicates that in each step, Algorithm 2 discovers a pair of keys x and y, such that it is guaranteed that the neighbor list NL does not contain keys in $I(x, y)$. In the following, we will show that the observation holds for all steps and utilize it to derive the complexity of Algorithm 2.

D. Analysis

In this section, we first show that at least $2n$ queries are needed to guarantee that an n-elemental neighbor list is retrieved, regardless of the choice of spoofed keys. Secondly, we analyze Algorithm 2 and show that it indeed terminates in at most $2n$ steps. Our results are mainly concerned with the *worst-case complexity*, defined as the maximal cost, i.e., number of queries in our cases, required by any input, i.e., neighbor lists in our case, for the algorithm to terminate.

More precisely, we show the optimality of our proposed algorithm as follows:

1) The complexity of the problem to obtain a provable complete neighbor list is $2n$, i.e., there exist some neighbor lists for which at least $2n$ keys need to be spoofed regardless of the algorithm for choosing these keys.
2) Our proposed algorithm needs at most $2n$ steps to obtain a provable complete neighbor list, and is hence optimal with regard to the worst-case complexity.

By the above, we only show that the algorithm achieves optimal performance with regard to the worst-case complexity, which is not necessarily optimal for all inputs. However, we later show that while there are lists that require less steps, most neighbor lists actually require at least $2n$ spoofed keys, regardless of how these keys are chosen. Hence, our performance bound is indeed of practical relevance. In order to present a more general result, we consider b-bit keys rather than the concrete value of $b = 160$ used in *P2P Zeus*.

Proposition III.3. *There exist neighbor lists* NL *with* n *distinct keys, such that the number of queries needed to guarantee the complete retrieval of* NL *is at least* $2n$ *for any choice of spoofed keys.*

Due to space constraints, the proof is presented in Appendix A. The idea of the proof is to divide the set $I(id_j, id_{((j+1) \bmod n)})$, id_j being the j-th smallest key in NL, into two sets F_{j-} and F_{j+}, such that keys in F_{j-} are closer to id_j and keys in F_{j+} closer to id_{j+1}. We then show that each non-empty F_{j-} or F_{j+} requires at least one query, adding up to a total of $2n$ queries.

We have shown that neighbor lists exist such that $2n$ spoofed keys are needed by any algorithm. However, as indicated by Example III.2, it is possible that a neighbor list can be discovered using less than $2n$ queries. Moreover, there exist an edge-case where total entries in neighbor lists might be empty or shorter than the minimum reply size, i.e., $|\mathrm{NL}| < l$. For these we only need exactly one query to retrieve the full list. The existence of such examples raises the question if the worst-case complexity is a suitable measure, or if it is only relevant for few constructed examples. However, Proposition III.4 and the subsequent calculation of Eq. 2 for realistic botnet sizes m show that the vast majority of neighbor lists require at least $2n$ spoofed keys for crawling, such that our algorithm is not only optimal with regard to the worst-case complexity, but also for nearly all inputs.

Proposition III.4. *The probability that the guaranteed retrieval of an* n*-elemental neighbor list in a P2P Zeus botnet with* m *bots and* b*-bit keys requires less than* $2n$ *spoofed keys*

is at most

$$\frac{m(m-1)}{2^{b+1}}\left(3+4(b-1)\ln 2\right). \quad (2)$$

Due to space constraints, we provide the proof in the Appendix C. The proof idea is to provide a bound on the probability that any F_{j-} or F_{j+}, as defined in the proof of Proposition III.3, is empty. The result is obtained using basic probability theory.

In *P2P Zeus*, typically, b = 160-bit keys are used, while the size of most such botnets is usually not bigger than several ten thousands. For such parameters, Eq. 2 is negligible, being less than 10^{-33}, so that indeed the vast majority of neighbor lists require $2n$ crawl requests to reveal all entries. A more detailed discussion of Eq. 2 can be found in Appendix B.

Now, we consider Algorithm 2 and show that its complexity is indeed $2n$.

Proposition III.5. *Algorithm 2 guarantees the complete retrieval of any n-elemental neighbor list* NL *and requires at most $2n$ queries.*

Proof: Denote the keys in NL by $id_0, \ldots, id_{n-1}$, sorted in ascending order. In the first step of Algorithm 2 (Lines 2 - 7), two keys are spoofed to guarantee that NL does not contain keys in $I(id_{n-1}, id_0)$. In each iteration of the loop, Algorithm 2 requires at most two queries with spoofed keys s_1 and $s_2 = s_1 + 1$ to guarantee that NL does not contain keys in $I(id_j, id_{j+1})$ for some $0 \leq j \leq n-2$ (Lines 10-26), id_j being the closest key to s_1 and id_{j+1} being the closest key to s_2. So, $n-1$ iterations of the loop are required to provably retrieve the neighbor list NL, since each pair (id_j, id_{j+1}) is only considered once. Hence, the two initial queries in addition to the maximum of $2(n-1)$ queries executed during the loop results in an upper bound of $2n$ on the number of queries. ■

We have now shown that Algorithm 2 achieves the lowest possible worst-case complexity. Note that the number of required steps can be much lower: If the keys contained in the list are consecutive, i.e., $id_{j+1} = id_j + 1$, our algorithm terminates in n steps. Furthermore, there are some neighbor lists, for which we cannot achieve the lowest possible number of steps, e.g., for a neighbor list consisting of only $\mathbf{0}(b)$ and $\mathbf{1}(b)$, Algorithm 2 requires four steps, but a variant of Algorithm 2 that starts using $0||\mathbf{1}(b-1)$ and $1||\mathbf{0}(b-1)$ for spoofing terminates within two steps. The example clearly shows that for any choice of initially spoofed keys and hence for any algorithm, there is some neighbor list for which the minimal number of steps is not achieved. However, Proposition III.4 shows that we are optimal for the vast majority of cases.

In the following, we present the evaluation results of ZEUSMILKER on a real *P2P Zeus* dataset along with some additional analysis.

E. Evaluation

In this section, we first describe the details of the used dataset and explain our experimental setup. Then, we present the evaluation results of our proposed ZEUSMILKER algorithm in comparison to other existing techniques. In more detail, we specifically investigate the impact of different neighbor list sizes n and different numbers of returned entries l per query on the efficiency of the different crawling techniques. Moreover, we also evaluate their efficiency in the presence of neighbor lists with different key distributions.

1) Experimental setup: In the following, we first describe the used dataset. Then we describe our simulation model and the metrics we used in the evaluation. Finally, we summarize our experiments with our initial expectations on them.

a) Dataset: For our evaluation, we utilized a real-world *P2P Zeus* dataset that consists of crawled information collected in a duration of approximately five hours from the botnet on 25th April 2013. The sanitized dataset contains information of 900 bots that have between 10 to 70 entries in their respective neighbor list. The median of the dataset is 34 entries with a standard deviation of 18.37. This dataset is provided as input data for our simulation model.

b) Simulation model: We implemented the *P2P Zeus* membership management protocols in *OMNeT++*[1] by making use of *OverSim*[2] [15] as our simulation framework. Our implementation includes the neighbor list restriction mechanism as described in Algorithm 1. For the generation of random keys within *OverSim*, we utilized the *OverlayKey* class to generate keys following a uniform distribution. For each iteration of the experiment, data for each bot in the simulation is uniformly selected at random from the described dataset depending on the investigated neighbor list size. The bots are then assigned the selected key and have their neighbor list filled with the associated neighbor list entries. Since the order of entries in a *P2P Zeus* bot's neighbor list is non-deterministic, we applied a random permutation to the contained entries before storing them in the neighbor list. To evaluate the efficiency of the neighbor list restriction mechanism in *P2P Zeus*, we implemented the following algorithms in our simulation framework:

- **ZEUSMILKER** is our proposed algorithm for strategically spoofing keys to *milk* all entries from a bot's neighbor list implemented as per Algorithm 2.
- ***Random*** is the only known algorithm used for monitoring *P2P Zeus* [4]. The spoofed keys are 160-bit keys generated uniformly at random.
- ***BinaryHalving*** spoofs keys by halving the ID space in the manner of a binary search algorithm. For each iteration of the algorithm, two keys are derived between two previously crawled keys. This is repeated until the maximum number of permitted requests is reached. For that, *BinaryHalving* initially spoofs with $\mathbf{0}(b)$ and $\mathbf{1}(b)$, and adds the pair $(\mathbf{0}(b), \mathbf{1}(b))$ to a FIFO queue Q. Then it executes the following statement T times:
  1) Remove the head (K_1, K_2) of Q and determine the keys $h_1 = \lfloor \frac{K_1+K_2}{2} \rfloor$ and $h_2 = h_1 + 1$,
  2) Crawl using spoofed keys h_1 and h_2, and
  3) Add (K_1, h_1) and (h_2, K_2) to Q.

c) Metrics: We measured the success of anti-crawling countermeasures and the performance of ZEUSMILKER by the *discovery ratio*. It is defined as the fraction of a neighbor list that is retrieved during crawling. Hence, the discovery ratio is an assessment for both the efficiency of the crawling algorithm as well as the effectiveness of the botnet's countermeasures, allowing the comparison of different crawling and anti-monitoring strategies. For each parameter set, the results are

[1] http://www.omnetpp.org
[2] http://www.oversim.org

averaged over 50 independent trials with confidence intervals of 95%. Furthermore, for each iteration of our experiments, a unique seed value has been used to initialize the simulation models and to choose a random node from the dataset. In all our experiments, we limit the maximum number of requests to $2n$ requests, in agreement with the worst-case complexity for retrieving a complete neighbor list (see Proposition III.3).

d) Influence of n and l: One of the countermeasures to hinder successful botnet monitoring is to restrict the number of entries that are returned after receiving neighbor list request, e.g., *Sality* [2] returns only a single entry for each request. For this reason, we investigate the influence of different neighbor list sizes n and neighbor list return sizes l. In this investigation, we expect ZEUSMILKER to successfully obtain all entries with at most $2n$ requests in every scenario as shown in Proposition III.5. Meanwhile, *Random* and *BinaryHalving* are expected to miss some entries. The *Random* crawling strategy retrieves a randomized set of entries and has a high probability of missing one or more keys. *BinaryHalving*, in contrast, divides the search space strategically, but does not make use of knowledge gained in previous steps and as such may continue to query regions with few or no keys intensively. Furthermore, we expect the performance of all algorithms to increase with increasing l, because more keys are discovered in each step. The increase should be particularly strong for *Random* as the probability to be successful when spoofing randomly is highly dependent on the number of trials.

e) Influence of different key distributions: The distribution of keys within a real world *P2P Zeus* bot's neighbor list is biased to the key of the bot itself [4]. However, due to the *P2P Zeus* neighbor list return mechanism, different key distributions can influence the number of unique entries that are able to be retrieved using choices of spoofed keys. Therefore, we have further analyzed the performance of the algorithms under two additional types of key distributions within a neighbor list:

- ***Random Distribution***: A node's neighbor list contains only randomly generated keys.
- ***Consecutive Entries***: A node's neighbor list contains only consecutive keys, e.g., $\mathrm{K}_{j+1} = \mathrm{K}_j + 1 \mod 2^{160}$.

We expect ZEUSMILKER to retrieve all entries within a bot's neighbor list independent of the chosen distribution. However, we even expect it to retrieve *Consecutive Entries* in only n requests instead of $2n$, as it is not necessary to check for additional keys between two neighboring keys. In contrast, *Random* and *BinaryHalving* are expected to require more crawling requests in this setting. Especially, *BinaryHalving* is expected to perform worst, as it spoofs many keys that yield no new knowledge in the *Consecutive Entries* setting. However, in the *Random Distribution* setting, both are expected to be closer, but still inferior to the crawling performance of ZEUSMILKER, as a result of the uniform key distribution.

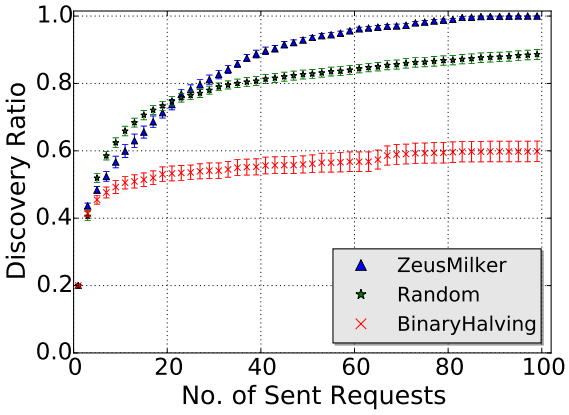
2) Results: In the following, we summarize our evaluation findings on the impact of l, n, and on the impact of the assumed key distribution on the three different crawling algorithms.

a) Impact of the size of the returned neighbor lists l: First, we discuss the impact of the size of the returned neighbor list l. Figure 2(a) summarizes the discovery ratio for a default parameter setting of a *P2P Zeus* botnet with $n = 50$ and $l = 10$ in dependence on the number of requests for all three crawling strategies. As can be seen, ZEUSMILKER is able to successfully retrieve all entries in a bot's neighbor list within 100 requests as guaranteed by Proposition III.5. At the same time, *Random* discovers only 92% and *BinaryHalving* even only 53% of all entries in a bot's neighbor list. Thus, the results confirm our expectation that *BinaryHalving* is not suitable for such biased neighbor lists. *BinaryHalving* performs poorly because of retrieving many duplicate entries as a result of spoofing keys within a range of the key space that provides no additional new information. For all algorithms, the number of initially retrieved entries increases fast with only a few queries. Later on, when only few keys are left undiscovered, the slope of the performance curve decreases. Note that during the first few queries, the *Random* crawling algorithm even manages to discover more unique keys than ZEUSMILKER. A potential reason for the initially weaker performance of ZEUSMILKER is the choice of the two spoofed keys s_1, s_2 (see Eq. 1), which are potentially very close and hence can result in returned sets with a high overlap. However, ZEUSMILKER is clearly superior to *Random* and *BinaryHalving* in discovering larger portions or even the complete neighbor list.

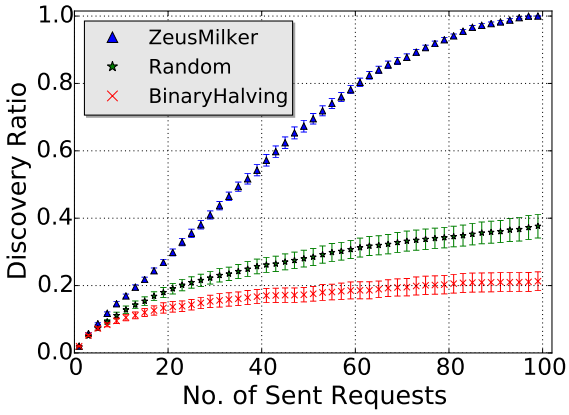
Figure 2(b) shows the discovery ratio in dependence on the number of crawling requests for $n = 50$ and $l = 1$. As can be seen, ZEUSMILKER still retrieves all entries within the predicted 100 requests, though at a lower speed than for $l = 10$. As only one entry per request can be obtained, the number of retrieved keys initially increases linearly and then converges slowly to a discovery ratio of 1. The decrease in performance is more apparent for *Random* and *BinaryHalving*: The discovery ratio for both approaches decreases drastically compared to $l = 10$, to 19% for *BinaryHalving* and 37% for *Random* at 100 requests. A more detailed analysis of the impact of l is given in Figure 2(c) showing the discovery ratio in dependence on l for $n = 50$. ZEUSMILKER successfully obtains all neighbor entries within $2n = 100$ queries independent of l, whereas the discovery ratios after $2n$ queries of the other two strategies are significantly affected by l. Since both algorithms are unable to strategically spoof keys, the fraction of retrieved keys drastically decreases when the number of returned keys l is reduced. Hence, the results of this analysis match our initial expectation that smaller values of l restrict the amount of new knowledge the crawling algorithms could obtain. However, since ZEUSMILKER is able to strategically spoof keys to discover all entries in a neighbor list, its ability to retrieve the complete list remains unaffected by different values of l.

b) Impact of the size of the neighbor lists n: Next, we analyze the impact of the size of the neighbor list n on the crawling performance. Figure 2(d) shows the discovery ratio of the different algorithms in dependence on n for $l = 10$. Independent of n, ZEUSMILKER successfully discovers all nodes in a neighbor list. In contrast, the performance of *Random* slowly decreases with increasing n, because it is harder to discover large sets simply by random trials than on smaller sets. The slight decrease in performance of *BinaryHalving* is not significant.

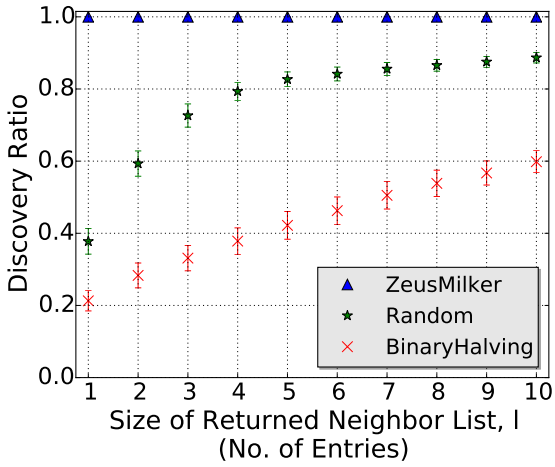
c) Influence of different key distributions: Apart from n and l, different key distributions in neighbor lists can also influence the performance of crawling algorithms. Figure 3(a) shows the discovery ratio in dependence on the number of requests for all three crawling strategies in the *Random Distribution* setting. As expected, ZEUSMILKER is able to obtain all entries with at most $2n = 100$ requests, whereas



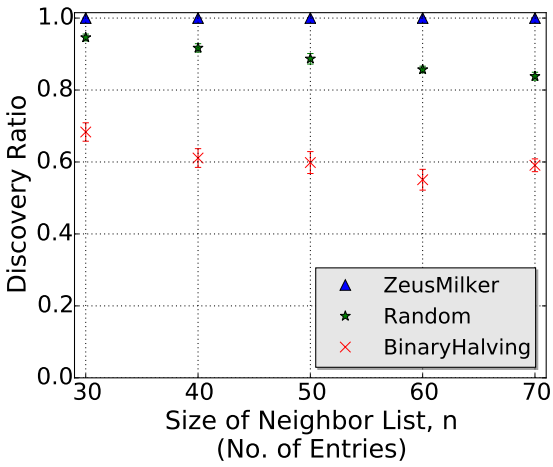
(a) *P2P Zeus* Milking Analysis (**n = 50**, **l = 10**)



(b) *P2P Zeus* Milking Analysis (**n = 50**, **l = 1**)

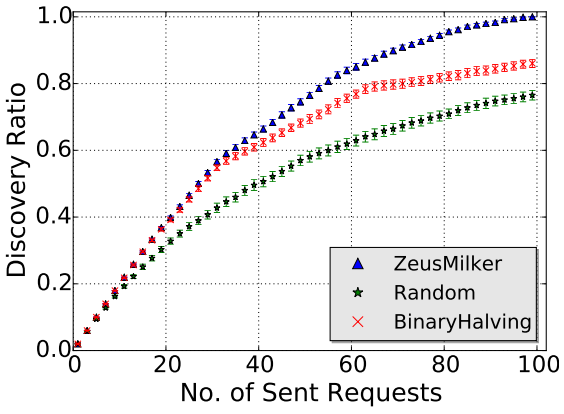


(c) *P2P Zeus* Milking Analysis (**n = 50**)

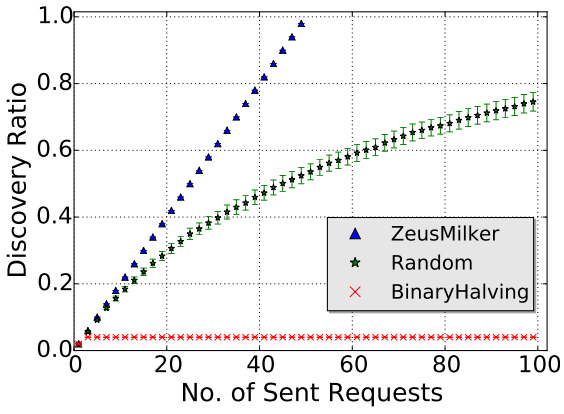


(d) *P2P Zeus* Milking Analysis (**l = 10**)

Fig. 2. Performance analysis of ZEUSMILKER, *Random*, and *BinaryHalving* for various neighbor list sizes n and returned neighbor list sizes l



(a) *P2P Zeus* Milking Analysis for **Random Distribution**



(b) *P2P Zeus* Milking Analysis for **Consecutive Entries**

Fig. 3. Performance analysis of ZEUSMILKER, *Random*, and *BinaryHalving* for different key distributions in neighbor lists (**n = 50**, **l = 1**)

Random and *BinaryHalving* only discover about 80% and 90% of all neighbor list entries, respectively. Both strategies perform considerably better than for the real-world data set, increasing their discovery ratio by more than a factor of 2 and 4, respectively. This improved performance arises from the well distributed keys, resulting in less duplicates during successive crawling attempts. As expected, *BinaryHalving* also performs much better than *Random* when the uniform key

distribution assumed by *BinaryHalving* is indeed given.

The inability of *BinaryHalving* to deal with non-uniform key distributions becomes evident when considering *Consecutive Entries*. *BinaryHalving* mostly discovers only two keys, i.e., discovery ratio is about 4%. A potential reason is the repeated spoofing of keys at distances away from all keys in the neighbor list. As a result, the same two keys are returned repetitively. ZEUSMILKER, in contrast, is able to successfully discover all entries with only n requests instead of $2n$, because the sets $I(K_j, K_{j+1})$ are empty, so that no additional n keys need to be spoofed to verify that $I(K_j, K_{j+1}) \cap \mathrm{NL} = \varnothing$. In contrast, the performance of the *Random* crawling is similar to its performance when considering randomly distributed keys.

d) Evaluation Summary: We have extensively evaluated the performance of ZEUSMILKER. As guaranteed by the proof in Section III-D, ZEUSMILKER always retrieves the complete neighbor list in at most $2n$ steps. Thus, it significantly outperforms existing crawling strategies like *Random* and *BinaryHalving*. Moreover, the performance of ZEUSMILKER is influenced neither by the size of a bot's neighbor list n nor by the number of returned entries l during each crawling step. Different key distributions have little to no effect on *Random*, whereas *BinaryHalving* is heavily influenced. ZEUSMILKER, in contrast, even performs significantly better on a consecutive distribution as only n rather than $2n$ requests are required to retrieve the complete neighbor list.

Next, to anticipate the next step of the botmasters, we present some advanced botnet countermeasures.

IV. COUNTERMEASURES

As algorithms like ZEUSMILKER circumvent the anti-crawling strategies of botnets, we anticipate the retaliation of the botmasters. For that, we introduce and evaluate more advanced countermeasures against crawling in this section.

A. Requirements of Crawling Countermeasures

There are three basic requirements:

1) Restricted Neighbor List: Any countermeasure should ensure that only limited neighbor entries are returned per query to prevent easy exposure of entries in the neighbor list.

2) Inference Prevention: To prevent an attacker from inferring about the entries in a neighbor list, the requester should not be able to influence the decision of which subset of the entries are returned. In addition, requesters should always be returned the same subset of entries whenever possible to restrict their view on the neighbor list of other bots. In this manner, analysts are prevented from obtaining an accurate snapshot of the whole topology, making takedown attempts unlikely to succeed.

3) Connectivity Preserving: The connectivity and hence the resilience to node failures and takedowns of the botnet should not be negatively affected, i.e., each benign bot should be equally often contained in the neighbor lists of other bots. Hence, the algorithm should not bias replies towards specific keys for all requesters. Such a bias might lead to an uneven degree distribution, which drastically reduces the resilience of the topology towards targeted node takedowns [16].

B. Anti-Crawling countermeasures

We implemented two countermeasures to improve the existing *P2P Zeus* neighbor list restriction mechanism according to our previous requirements, along with another countermeasure that simply returns random nodes from the neighbor list. We briefly present the countermeasures and discuss the expectations in the following.

1) Random Node Return: In *Sality* [2], bots return exactly one randomly chosen entry from their respective neighbor lists to the requesting bot. Hence, the requesting bot has no influence on the returned entries at all. However, by returning different sets for each query, a considerable portion of the neighbor list can be easily obtained using repeated queries. Hence, both *Random* and *BinaryHalving* are expected to be comparable in their discovery ratio.

2) Bit-XOR+: This countermeasure adds additional randomness at the side of the recipient of a neighbor list request. The recipient generates a random key uniformly for each IP address it receives a request from and stores it. This key is then *XOR*-ed with the key of the requesting node and the resulting key is then used as an input for Algorithm 1 to return the neighbor entries. Hence, the set of keys that is returned is now biased towards the new *XOR*-ed key and an attacker loses its ability to strategically spoof keys. As a result, the discovery ratio of *Random* and *BinaryHalving* should both be the same as in *P2P Zeus*. ZEUSMILKER is expected to perform worse because of the introduced randomness, as it might incorrectly assume that all keys have been retrieved and stops querying. By including a randomly generated key into the selection process, each entry in the neighbor list has the equal likelihood to be returned, such that *Bit-XOR+* is expected not to negatively affect the connectivity.

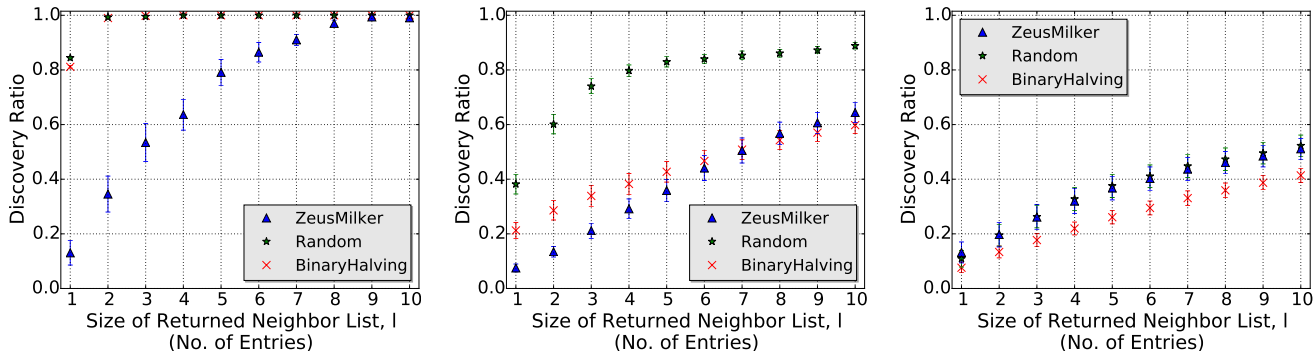
3) Bit-AND: Bit-AND is a variation of the *Bit-XOR+* countermeasure that executes a bit-wise AND operation between the stored key and the requesters key before using the resulting key to return neighbor list entries. However, due to the nature of the AND operation whereupon each bit of the resulting key has a tendency to be 0 with a probability $3/4$, the set of returned keys for *Bit-AND* is likely to be biased towards keys starting with 0s. On the one hand, such a bias considerably decreases the performance of all observed crawling strategies, because the returned sets are expected to have a larger overlap in contrast to uniformly selected sets. On the other hand, keys starting with 1s are expected to be present in fewer neighbor lists, potentially damaging the connectivity and thus the resilience of the botnet. Therefore, while *Bit-AND* is expected to achieve the best performance out of the three countermeasures, its disadvantages likely outweigh its benefits.

C. Evaluation Results

In the following, we present the analysis results of our countermeasures in Figure 4 which shows the discovery ratio of the different crawling algorithms in dependence on the different neighbor list return sizes.

The *Random Node Return* analysis in Figure 4(a) indicates the inefficiency of this countermeasure in restricting the information gained by a crawler. Both *Random* and *BinaryHalving* were able to retrieve more than 80% of a bot's neighbor list in all parameter settings after 100 requests. However, ZEUSMILKER performed poorly as expected due to the inherent incorrect assumptions made on the returned keys that led the algorithm falsely assuming there are no more keys left undiscovered.

Our results for the *Bit-XOR+* countermeasure indicate that



(a) **Random Node Return** (b) **Bit-XOR+** (c) **Bit-AND**

Fig. 4. Performance analysis of ZEUSMILKER, *Random*, and *BinaryHalving* on the presence of different advanced countermeasures (**n = 50**)

Random performs best with an average of about 80% of nodes discovered for $l \geq 4$, as displayed in Figure 4(b). Hence, the performance of *Random* is largely not influenced by bits flipping, as can be seen from comparing Figure 4(b) and Figure 2(c), showing the performance of the crawling for the unaltered *P2P Zeus*. Although *BinaryHalving* initially performs better than ZEUSMILKER for $l \leq 7$, its performance degrades for $l > 7$, as a result of spoofing keys that yield more duplicate entries. However, ZEUSMILKER's strategy of deriving keys based on previous knowledge provides more randomness, i.e., variety of key prefixes, in the spoofed keys, hence obtains a slight improvement than *BinaryHalving* towards the end.

Bit-AND, as displayed in Figure 4(c), presents a better restriction mechanism than *Random Node Return* and *Bit-XOR+* as the discovery ratio of all crawling algorithms is kept below 50% for $l \leq 10$. The discovery ratio increases with the size of the returned neighbor lists l in a close to linearly manner. Although the poor performance of all strategies in terms of discovery ratio indicates the effectiveness of this countermeasure, the bias resulting from this strategy may negatively affect the robustness of the resulting overlay.

Although *Bit-AND* is more effective in hampering the performance of the crawling algorithms, the introduced bias in the returned keys contradicts to our requirements. While the *Random Node Return* strategy is able to prevent an attacker to provably discover all entries, the amount of information exposed to crawlers is significantly high. Thus, from the presented results above, we can conclude that *Bit-XOR+* is the best suitable countermeasure for future P2P botnets that not only prevents any strategic *milking* attempts, i.e., cannot derive any information, but is also able to ensure a more resilient overlay due to the randomness that results from its design.

V. CONCLUSION

In this work, we proposed ZEUSMILKER, an algorithm to circumvent the neighbor list restriction mechanism of *P2P Zeus*. We evaluated the performance of our proposal theoretically to prove that ZEUSMILKER is capable to provably obtain the complete neighbor list of a bot of size n in at most $2n$ crawling steps. Our extensive simulation results also indicate that ZEUSMILKER performs significantly better than existing crawling mechanisms on *P2P Zeus*. As seen in the past, botmasters are continuously adapting their botnets against new threats and challenges from the defenders. Hence, to anticipate future countermeasures by the botmasters, we have evaluated the effectiveness of several anti-crawling techniques against ZEUSMILKER. The suggested *Bit-XOR+* countermeasure prevents any attempts to strategically retrieve entries from a given neighbor list by introducing additional randomness at the side of the recipient of a neighbor list request.

As future work, we intend to investigate the impact of these advanced anti-crawling countermeasures on the structure of the resulting botnet overlay. This would allow us to better understand the impact of design choices across different botnets and to predict the design of future botnets.

REFERENCES

[1] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos, "Highly Resilient Peer-to-Peer Botnets Are Here: An Analysis of Gameover Zeus," in *Proceedings of the 8th IEEE International Conference on Malicious and Unwanted Software*, 2013.

[2] N. Falliere, "Sality: Story of a peer-to-peer viral network," Symantec Corporation, Tech. Rep., 2011.

[3] J. Wyke, "The ZeroAccess Botnet – Mining and Fraud for Massive Financial Gain," Sophos, Tech. Rep. September, 2012.

[4] C. Rossow, D. Andriesse, T. Werner, B. Stone-gross, D. Plohmann, C. J. Dietrich, H. Bos, and D. Secureworks, "P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets," in *Symposium on Security & Privacy*. IEEE, 2013.

[5] J. Kang and J.-Y. Zhang, "Application Entropy Theory to Detect New Peer-to-Peer Botnet with Multi-chart CUSUM," in *Second International Symposium on Electronic Commerce and Security*. IEEE, 2009.

[6] 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 *16th Conference on Computer and communications security*. ACM, 2009.

[7] S. Karuppayah, M. Fischer, C. Rossow, and M. Muhlhauser, "On Advanced Monitoring in Resilient and Unstructured P2P Botnets," in *International Conference on Communications*. IEEE, 2014.

[8] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Peer-to-Peer Systems*, ser. LNCS. Springer Berlin Heidelberg, 2002, vol. 2429.

[9] C. Kanich, K. Levchenko, and B. Enright, "The Heisenbot Uncertainty Problem: Challenges in Separating Bots from Chaff." *LEET*, 2008.

[10] 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." *LEET*, 2008.

[11] G. Starnberger, C. Kruegel, and E. Kirda, "Overbot: a botnet protocol based on Kademlia," in *4th International Conference on Security and Privacy in Communication Networks*. ACM, 2008.

[12] G. Yan, S. Chen, and S. Eidenbenz, "RatBot: Anti-enumeration Peer-to-Peer Botnets," in *Information Security*, ser. LNCS. Springer Berlin Heidelberg, 2011, vol. 7001.

[13] G. Yan, D. T. Ha, and S. Eidenbenz, "AntBot: Anti-pollution peer-to-peer botnets," *Computer Networks*, vol. 55, no. 8, Jun. 2011.

[14] R. Hund, M. Hamann, and T. Holz, "Towards Next-Generation Botnets," in *European Conf. on Computer Network Defense*. IEEE, Dec. 2008.

[15] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in *Global Internet Symposium*. IEEE, May 2007.

[16] R. Cohen, K. Erez, D. Ben-Avraham, and S. Havlin, "Breakdown of the Internet under Intentional Attack," *Physical Review Letters*, vol. 86, no. 16, Apr. 2001.

APPENDIX A
PROOF OF PROPOSITION III.3

Proof: We enumerate the keys id_j in NL, such that $id_j < id_{j+1}$ for $j = 0 \dots n-2$. For $1 \le j \le n-1$, denote the common prefix of id_{j-1} and id_j by $cp_j = cp(id_j, id_{j+1})$. Let $s_0 = \mathbf{0}(b)$ and $s_j = cp_j \| 1 \| \mathbf{0}(b - |cp_j| - 1)$ be the smallest key in $I(id_j, id_{j+1 \bmod n})$ that is closer to id_j than to $id_{j-1 \bmod n}$ with regard to the XOR distance (see Eq. 1 for more detailed explanation on why this is the case). In the following, we consider the sets $F_{j-} = I(id_{j-1 \bmod n}, s_j)$ and $F_{j+} = I(s_j - 1, id_j)$ for $j = 0 \dots n-1$, and explain that at least one spoofed key has to be chosen in F_{j-} and F_{j+} each for all j. First note that by construction, the $2n$ sets $F_{j-} \cup F_{j+}$ for $j = 0 \dots n-1$ are all disjunct. For non-empty F_{j*} with $0 \le j \le n-1$ and $* \in \{+,-\}$, consider an arbitrary key $x \in F_{j*}$. If x is the first element in a neighbor list $\mathrm{NL}' = \mathrm{NL} \cup \{x\}$, it is only returned if a spoofed key s with $\mathrm{XOR}(s,x) < \mathrm{XOR}(id_i, x)$ for all $0 \le i \le n-1$ is used. However, by the construction of the sets F_{j*}, all such keys are contained in F_{j*}. Thus, at least one query is required for each non-empty set F_{j*}. Hence, for all neighbor lists NL with only non-empty F_{j*}, $2n$ queries are required. Such lists exists: An example for such a neighbor list is given by $id_j = j \cdot 16 + 1$, i.e. the first n hexadecimal numbers ending in 1. Then $F_{j+} = \{j \cdot 16\}$ and $F_{j-} = \{j \cdot 16 + 2, \dots, id_{(j+1) \bmod n}\}$ for all j. So, a lower bound on the worst-case complexity of an algorithm for guaranteed retrieval of neighbor lists is indeed $2n$. ∎

APPENDIX B
PROBABILITY OF NON-OPTIMAL PERFORMANCE

Figure 5 gives an upper bound on the probability that a neighbor list in a network of m bots is not retrieved by ZEUSMILKER at the optimal cost for $m \le 1,000,000$. The probability is computed based on Eq. 2.

APPENDIX C
PROOF OF PROPOSITION III.4

Proof: In this proof, we use $P(A)$ to denote the probability of an event A, $P(A|B)$ for the probability of A conditioned on B, and $\varnothing$ to denote the empty set. The proof of Proposition III.3 gives a precise description of neighbor lists requiring $2n$ spoofed keys, stating that the bound holds if all sets F_{j-} and F_{j+} are non-empty. We now give an upper bound on the likelihood that an empty F_{j-} or F_{j+} exists. Because the distribution of keys in a neighbor list is unknown, we obtain an upper bound on the probability that for any pair (x,y) of keys in the network, the set $I(x,y)$ of keys between them is empty.

Consider any two keys x and y: Recall that $I(x,y)$ denotes the set of keys between x and y. Choose x and y such that $|I(x,y)| \le |I(y,x)|$, i.e., we consider the shortest segment on

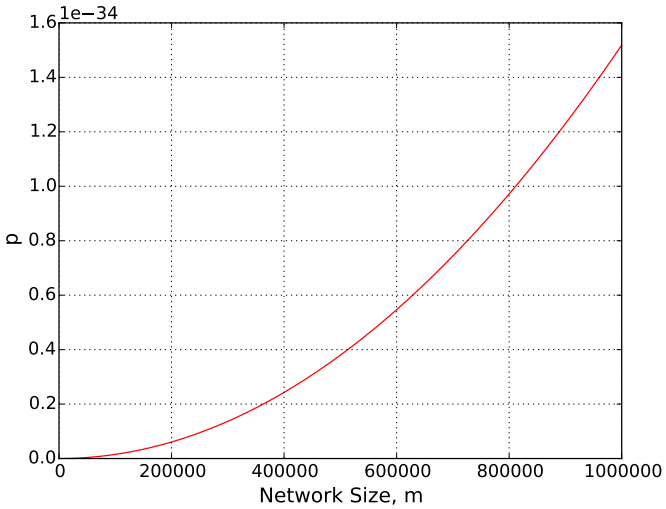


Fig. 5. Quantity in Eq. 2 for **b=160** bits

a ring of length 2^b between x and y. The size of $I(x,y)$ is then given by $|I(x,y)| = \min\{|y-x|, 2^b - |y-x|\} - 1$ Furthermore, let F_- and F_+ be the keys in $I(x,y)$ closer to x and y with regard to the XOR, respectively. If $|I(x,y)| \le 1$, either F_- or F_+ is empty. Otherwise, there are $|I(x,y)| + 1$ possibilities that the keys in $I(x,y)$ could be divided between F_- and F_+, and only for two of them F_- or F_+ is empty, namely if either $x = cp(x,y) \| 0 \| \mathbf{1}(b - |cp(x,y)| - 1)$ or $y = cp(x,y) \| 1 \| \mathbf{0}(b - |cp(x,y)| - 1)$. Hence F_- or F_+ is empty with probability $\frac{2}{|I(x,y)|+1}$. Let D denote the random variable giving $|I(x,y)| + 1$ for two uniformly chosen keys. The probability that D attains the value d is

$$P(D = d) = \begin{cases} \frac{1}{2^b}, & d \in \{0, 2^{b-1}\} \\ \frac{2}{2^b} & 1 \le d \le 2^{b-1} - 1, \end{cases}$$

so that F_- or F_+ is empty with probability

$$\begin{aligned} P(F_- = \varnothing \cup F_+ = \varnothing) &= \sum_{d=0}^{2^{b-1}} P(D = d) P(F_- = \varnothing \cup F_+ = \varnothing | D = d) \\ &= \frac{1}{2^b} \left(3 + \sum_{d=2}^{2^{b-1}-1} \frac{4}{d} + \frac{2}{2^b} \right) \qquad (3) \\ &\le \frac{1}{2^b} \left(3 + 4(b-1) \ln 2 \right). \end{aligned}$$

The last step follows because for the harmonic series $\sum_{d=1}^{m} \frac{1}{d} \approx \ln m + \rho$ for the Euler-Mascheroni constant $\rho = 0.577\ldots$. Note that the keys in a neighbor list are not distributed uniformly, but are usually close to the node's own key. However, the keys of the m nodes in the network are selected uniformly at random. We hence consider the probability that for none of the $m(m-1)/2$ pairs of keys, the set of keys closer to one of them is empty. An upper bound on the desired probability is obtained by a union bound using Eq. 3

$$P\left(\bigcup_j F_{j-} = \varnothing \cup F_{j+} = \varnothing \right) \le \frac{m(m-1)}{2^{b+1}} \left(3 + 4(b-1) \ln 2 \right).$$

∎