

# MUSE: Towards Robust and Stealthy Mobile Botnets via Multiple Message Push Services

Wei Chen<sup>1</sup>, Xiapu Luo<sup>2,3</sup>, Chengyu Yin<sup>1</sup>, Bin Xiao<sup>2</sup>,  
Man Ho Au<sup>2</sup>, and Yajuan Tang<sup>4</sup>(✉)

<sup>1</sup> School of Computer,  
Nanjing University of Posts and Telecommunications, Nanjing, China

<sup>2</sup> Department of Computing,  
The Hong Kong Polytechnic University, Hung Hom, Hong Kong

<sup>3</sup> The Hong Kong Polytechnic University Shenzhen Research Institute,  
Shenzhen, China

<sup>4</sup> College of Engineering, Shantou University, Shantou, China  
yjtang@stu.edu.cn

**Abstract.** Exploiting unique features in mobile networks and smart-phones, mobile botnets pose a severe threat to mobile users, because smartphones have become an indispensable part of our daily lives and carried a lot of private information. However, existing mobile botnets usually rely on a single command and control channel (e.g., a push server or an SMS server) to disseminate commands, which can become the bottleneck or a single point of failure, without considering the robustness. In this paper, we propose MUSE, a novel multiple push service-based botnet, which can significantly outperform existing push-styled mobile botnets in terms of robustness, controllability, scalability, and stealthiness. Although the basic idea of using multiple push services seems straightforward, we explore the design space of exploiting such services and tackle several challenging issues to overcome the limitations of existing push-styled mobile botnets. We have implemented MUSE by exploiting ten popular push services and evaluated it through extensive experiments. The results demonstrate not only MUSE's feasibility but also its advantages, such as stealthiness, controllability etc.

## 1 Introduction

With the rapid advent of mobile Internet access, mobile devices are used by millions of users due to their portable Internet access and the increasing powerful computing capabilities. Botnets have also evolved from the traditional PC-based form to smartphones based form, because smartphones usually lack of sufficient security mechanisms and store a lot of private personal information. The mobile botnet is a collection of the compromised mobile devices that can remotely receive commands from botmaster through C&C channels. It can also upload the sensitive information in the compromised devices, thus bringing serious threats to the property, privacy, and security of smartphone users.

## 1.1 Current Mobile Botnets

In recent years, the mobile botnets have become an important threat to the cyber security with the development of mobile Internet. For example, in 2010, the GEINIMI botnet appeared on Android platform. Soon iKee.B was reported as the first botnet in jailbroken iPhones. In 2011, AnserverBot, an Android bot, began to use public blogs as C&C [24]. This approach was different from previous C&C channels, which usually depended on the traditional HTTP channel. The Bmaster botnet controlled thousands of Android devices [13]. By controlling lots of compromised mobile devices, the botmaster can harvest private information [25] and launch various malicious activities, such as DDoS attacks against a cellular network core [19].

Since message push services eliminate the direct communication between bots and botmaster and provide well-maintained infrastructures to deliver messages, recent studies suggested constructing push-styled mobile botnets [10, 23]. However, existing research has not fully explored the capability of push services for building advanced mobile botnet. For example, Zhao et al. proposed using Google’s push service (i.e., C2DM: Google’s Cloud to Device Messaging Service) to build mobile botnets targeting on Android platform [23]. However, the botnet in [23] is not robust enough because its bots could be identified through the same embedded API key. If the API key was blocked by the defender, the whole botnet would fail. Lee et al. designed Punobot [10], a mobile botnet built upon Google Cloud Message for Android (GCM), the new version of C2DM. Although Punobot avoids using the same API key in all bots by exploiting a vulnerability in GCM’s registration process, it still has the single-point-of-failure problem and does not explore the design space of employing push services for mobile botnet. In this paper, we propose MUSE, a novel push-styled mobile botnet exploiting multiple diverse push services (e.g., GCM, JPush, etc.) to provide better robustness, controllability, scalability, and stealthiness.

## 1.2 Goals and Challenges

In this paper, we explore the design space of mobile botnets based on message push services because such services provide great convenience and flexibility for bots to receive messages. Although a few studies proposed mobile botnets based on push services, none of them could achieve all of the following features:

1. **Robustness.** How to empower the botmaster to maintain the control of bots even after substantial push service accounts have been blocked by the defender? Some push services do not provide global services. If these services have taken down in certain area, how to keep the botnet survive?
2. **Controllability.** When multiple push servers are organized together, the complexity of controlling the botnet increases. How to allow the botmaster to quickly disseminate commands to bots?
3. **Scalability.** How to scale up the botnet when numerous new mobile bots join the botnet?

4. **Stealthiness.** How to prevent (or make it harder for) defenders from detecting bots via their communication traffic patterns? How to make botnet traffic similar to normal traffic?

We propose MUSE, a novel multiple push service-based botnet, which can significantly outperform existing push-styled mobile botnets in terms of robustness, controllability, scalability, and stealthiness. MUSE involves a set of practical methods to address these challenges. More precisely, besides using multiple push services to avoid the single point of failure, we propose a hybrid structure to connect bots, push services, and the botmaster together. Such structures can not only scale up the MUSE botnet but also make it very difficult to dismantle the botnet. We design a set of mechanisms to improve a MUSE botnet’s stealthiness and performance in terms of controllability and scalability. First, we use a dynamic weight round robin algorithm for push server selection. It can help botmaster avoid performing excessive operations on certain servers. This algorithm also ensures that the C&C traffic is distributed among different servers. Second, since push services only support text messages, we employ high-order mimic functions to transform binary data into English text. Third, the LEACH protocol [6] is applied for dynamically selecting servant bots to construct intermediate layer. Finally, when implement MUSE by integrating ten push services, we discuss incremental update techniques for Android bots and bi-direction communication channels with HTTP Restful API.

In summary, we make the following contributions:

- We exploit multiple push services to construct robust and stealthy mobile botnets. Our design, MUSE, avoids the single point of failure problem in mono-push-server structure and has flexible structures to improve the robustness and scalability.
- We propose a set of mechanisms to improve a MUSE botnet’s stealthiness and performance in terms of controllability and scalability. More precisely, at the botmaster side, MUSE disperses the botnet traffic among different push servers, which makes mobile botnet stealthy. MUSE uses three independent channels to enhanced upstream C&C channels. It is possible for all mobile bots to establish bi-direction communication channels.
- We have implemented MUSE employing ten popular push services and conducted extensive evaluations on it. The incremental update method is proposed to reduce update traffic consumption. The experimental results show that MUSE can effectively enhance mobile botnets’ stealthiness and performance.

### 1.3 Paper Organization

Section 2 reviews the related work. Section 3 introduces MUSE’s architecture. Section 4 discusses the design of MUSE, including commands dissemination and servant bots selection. Sections 5 and 6 present the implementation and the evaluations of MUSE, respectively. Section 7 concludes the paper with future work.

## 2 Related Work

Although botnets in wired network have been extensively examined in the past eight years [9, 16, 17], mobile botnets just emerged along with the rapid advent of mobile Internet and the prosperity of smartphones. Since mobile devices have lower computation ability and users are sensitive to resource consumption, researchers proposed mobile botnets with special functions and abilities on smartphones.

Pieterse and Olivier [15] designed a hybrid C&C structure for mobile botnet to avoid single point of failure. They used the short message service (SMS), HTTP and bluetooth to build communication channels. Singh et al. [18] used bluetooth as a C&C channel to send commands from one bot to another. This mobile botnet was limited by the radio range of Bluetooth. Anagnostopoulos et al. [1] introduced two botnet architectures that consist only of mobile devices. One applied mobile HTTP proxy to build C&C and the other used DNS protocol as a covert channel. Zeng et al. [22] proposed a P2P botnet structure using SMS. With the P2P topological structure, botmaster could reduce the number of SMS messages required for bots communications and decrease the time delay for delivering SMS commands. Hua et al. [7] used SMS to construct a mobile botnet and evaluated the performance in different types of network topology. Eslahi et al. [5] made a survey about the current available data set and samples of mobile botnets. They tried to implement a mobile botnet test bed to collect data for further research. According to the Kademlia P2P network structure, Mulliner and Seifert [14] utilized an SMS-HTTP hybrid approach to established a P2P mobile botnet, in which the C&C was divided into HTTP and SMS parts. Karim et al. [8] investigated mobile botnet attacks and compared existing mobile botnets on commercial as well as open source mobile operating system platforms. Cui et al. [4] presented the URL Flux-based mobile botnet model for Android. The botnet commands were hidden into specified pictures and uploaded to the blogs. Bot visited the corresponding blogs to extract the commands from pictures. They also proposed a Botnet Triple-Channel Model with three independent sub-channel to enhance the upstream channel [3]. BTM botnet had a high-performance upstream channel, which would be attractive for botmaster.

The research work most relevant to ours is push-styled mobile botnets proposed by Zhao et al. [23]. Such botnets exploited push services to establish C&C channel and the commands were disseminated by Google's C2DM. They showed that the message push service can deliver commands to bots timely and effectively. The botnet command dissemination can be hidden into the legitimate push traffic to some extent. Lee et al. [10] followed Shuang's work and proposed a mobile botnet which exploited GCM as its C&C channel, which reduced battery power consumption, traffic cost and money cost. These works depended on the reliability and availability of a single push server. If the defender blocks the suspicious botnet accounts on the push server, the botnet will be paralyzed. Some push services are not available in some regions, such as GCM in China. In these regions, the push service becomes the single point of failure in the botnet. In our previous work [2], we addressed the single point of failure problem by

using multiple push servers to substitute one single server. But we did not consider how to construct a sophisticated structure with the multiple push servers. The proposed method used KING algorithm to measure path delay between bots and push servers. Then bots were clustered into different groups by DBSCAN algorithm according to path delay. A static weight round-robin algorithm was applied to select a push server to disseminate commands for each bot group. The dynamic performance of push servers was not considered in the push server selection. In this paper, we propose a hybrid structure, which is more flexible and stealthy. The robustness of MUSE is mathematically analyzed. To support this hybrid structure, we discuss the servant bot selection and bot bi-direction communication channel. MUSE also involves a set of practical methods to implement the botnet. For example, incremental update is applied to reduce update traffic consumption and messages for command transmission are preprocessed to fulfill the requirement of push services that only support text messages.

### 3 Architecture of MUSE

In this section, we describe the architecture of MUSE starting from a basic scenario(i.e., Fig. 1a) where one push service is used to advanced ones including flat structure(i.e., Fig. 1b) and hybrid structure(i.e., Fig. 1c).

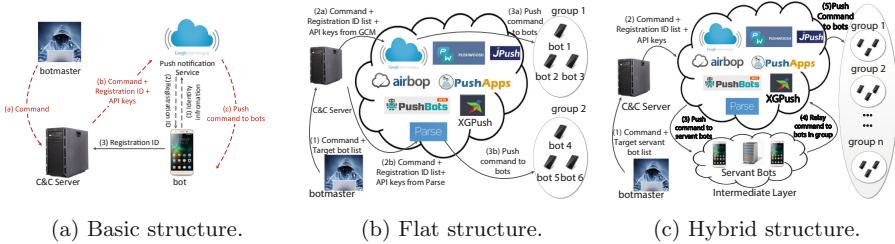


Fig. 1. MUSE's architecture.

#### 3.1 Basic Structure

As shown in Fig. 1a, the basic structure involves one push service. When a new bot joins the botnet, it first registers itself to the push service and then reports its registration ID, generated by the push service, to the botmaster. To disseminate commands to bots, the botmaster constructs a push request, which contains the command, the registration IDs, and required API Keys. After the botmaster sends the request to the push server, the push server will deliver the command to bots when they are online. Otherwise, the push server will store the command and send it later. The C2DM botnet [23] and the Punobot [10] can be considered as variants of the basic scenario.

The mobile botnet with push notification service eliminates the needs for bots to access the C&C server to retrieve the botnet commands intermittently. In addition, push-styled botnet avoids direct communication between the botmaster and the bot. However, the basic scenario has some limitations. The botnet depends on the availability of the push server. Some push services are not available in some regions, such as Google’s GCM in China. The push service may become the bottleneck for the whole botnet. What’s more, if the push server is blocked by the defender, the botnet will be paralyzed. MUSE can overcome these limitations by using the simple flat structure introduced in Sect. 3.2 or the advanced hybrid structure introduced in Sect. 3.3.

### 3.2 Flat Structure

As shown in Fig. 1b, the flat architecture enables the botmaster to send commands via different push servers to bots. Although push services may have diverse features, a botnet can exploit them by performing the following steps:

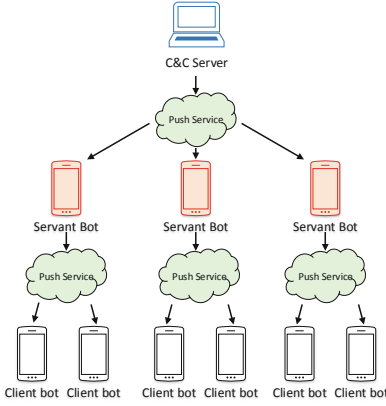
1. The bot obtains registration IDs from a push service. For example, by changing the identification information, such as the email address for GCM [10], a bot could have several registration IDs from one push service for improving its resiliency. Therefore, even if one registration ID is blocked by the defender, the bot can still receive messages from the push server using other registration IDs.
2. The bot sends all registration IDs to the botmaster. Since the amount of such information is small, a bot can easily deliver them to botmaster through email, SMS, and various covert channels [11].
3. The botmaster registers itself as one or more server applications to obtain accounts in each push service. After getting bots’ registration IDs, the botmaster can send push messages that contain commands to bots according to their registration IDs and the corresponding push service accounts.
4. When a push service receives messages from the botmaster, it will dispatch them to bots according to their registration IDs.

In the flat structure, a push service controls several groups of bots and each bot can receive messages from more than one push service. Therefore, even if one push service is down or bots associated with one push service are blocked, bots can still receive commands through other push services. Moreover, since the botnet traffic is distributed among multiple servers, MUSE introduces much less additional traffic to normal connections with push services than the basic scenario does, thus improving the stealthiness. Note that in the flat structure there are no connections among different push services. To further improve a MUSE botnet’s robustness, stealthiness, and flexibility, we propose a hybrid structure so that the botmaster can dynamically change the botnet’s structure (e.g., hierarchy structure, P2P structure, etc.).

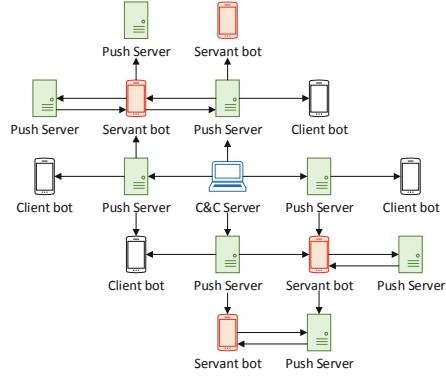
### 3.3 Hybrid Structure

As shown in Fig. 1c, the hybrid structure has one or more intermediate layer that includes a set of compromised smartphones or dedicated servers running emulator. We call them servant bots, and they usually have good capability, such as powerful CPU, stable network connection, and large bandwidth. Besides joining multiple push services to receive commands, servant bots can relay messages received from one push service to other push services through their APIs.

With the hybrid structure, the botmaster can easily re-organize the structure of bots. For example, the botmaster can construct a hierarchy structure by letting servant bots serve as internal nodes of a tree and normal bots as leaf nodes. Figure 2a shows the hierarchy structure, where C&C server becomes the root of the tree and servant bots work as internal nodes. All bots become the leaf nodes and internal nodes send messages to its children through push services.



(a) Hierarchy structure.



(b) P2P structure.

**Fig. 2.** Change to different structures.

The botmaster can also build a hybrid P2P botnet [20] where push services act as super nodes. As shown in Fig. 2b, servant bots make push services completely connected. A servant bot can connect to multiple push services and each push service can send commands to multiple client bots. When a new bot joins botnet, it can send registration information to multiple servants bots. With the P2P structure, the commands can spread over the botnet in a short time.

With different structures, a botnet becomes more robust, because bots have more approaches to receive commands, and more stealth, because bots can also easily change their traffic patterns.

### 3.4 Robustness Analysis

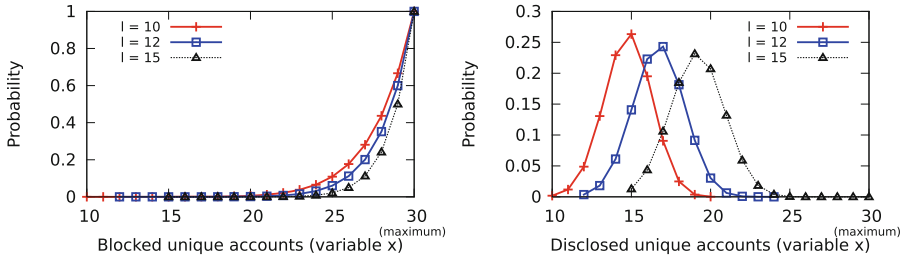
We conduct an analytical study on the robustness of botnet. Assume that there are  $n$  push servers and botmaster registers  $m$  accounts on each push server.

These accounts are shared by all bots. Each bot maintains an account list and suppose there are  $l$  accounts in each bot list. In MUSE's implementation, each bot randomly chooses  $l$  accounts from all shared  $n \times m$  accounts. The account list can be set in the manifest file in Android application. It is not difficult for bots to use different account lists in practice.

One defense method against MUSE is to catch some bots and retrieve the account information by reverse engineering bot malware. Then, they can block these accounts by informing push services providers. We assume that defenders catch  $b$  bots and get  $b \times l$  accounts information. Since the accounts are shared by bots, some accounts may be in more than one bots. We let  $x$  stand for the number of unique accounts.

Note that only when all accounts in a bot list are blocked by the defenders, this bot will lose the connection to the botmaster. If there is one account left, the bot can still receive information from the botmaster to update its account list. When  $x$  accounts are blocked, the probability of a bot being blocked is:

$$P(x) = \frac{C(x, l)}{C(n \times m, l)}, \quad x \in [l, \max(b \times l, m \times n)] \quad (1)$$



(a) Blocked probability with different accounts number. (b) Disclosed probability when bots are captured.

**Fig. 3.** Robustness mathematical analysis. (Color figure online)

Figure 3a shows the analytical result of botnet robustness. The parameters are set as  $n = 10, m = 3, b = 2$ . It shows that when the blocked account  $x$  is not close to maximum value, the botnet can still be safe. Moreover, if the botmaster uses more push servers ( $n$ ) and applies more accounts ( $m$ ), the botnet will be more robust.

Now we analyze how many unique accounts( $x$ ) appear in all caught accounts( $b \times l$ ). When  $b$  bots are caught,  $b \times l$  accounts on these bots are blocked. The number of unique accounts  $x$  is:

$$N(x) = C(m \times n, x) \sum_{k=0}^{x-1} (-1)^k C(x, k) (x - k)^{b \times l} \quad (2)$$



The probability of  $x$  unique accounts being caught among  $b \times l$  accounts is:

$$P_n(x) = \frac{N(x)}{\sum_{x=l}^{\min(b \times l, m \times n)} N(x)} \quad (3)$$

Figure 3b shows the probability of disclosing unique accounts when bots are captured. The parameters are set as  $n = 10, m = 3, b = 2$ . The result shows that it is not easy for defender to disclose all accounts when a small part of bots are captured.

## 4 Design of MUSE C&C

The C&C mechanism is the core part of the botnet. One task of C&C is to efficiently disseminate botnet commands to bots. It is also important to hide and protect command and control traffic to avoid detection. In this section, we describe the design of MUSE's C&C mechanism for command dissemination with the consideration of controllability and stealthiness.

To enhance the controllability, bots are initially categorized into different groups according to their locations. We propose a new algorithm to help the botmaster select a push server for sending commands to different bot groups. To scale up the botnet, a hybrid architecture has been introduced in Sect. 3.3. Here, we discuss how to select servant bots from bot groups to construct intermediate layer. Finally, we employ high-order mimic functions to transform binary data into English text because push services only support text messages.

### 4.1 Push Server Selection

Push servers usually provide services for global users and deploy distributed system of servers in multiple data centers across the Internet, which is similar to CDN (Content Distribution Network) technology. The goal of distributed deployment is to push content to end-users with high availability and high performance.

For individual smartphone, different push servers have different performance. To improve the controllability of a botnet, a push server with best performance is preferred to disseminate commands. In order to simplify command dissemination, the bots from the same region are categorized into the same group. The botmaster selects one push server for each group to send commands.

During push server selection, several factors should be considered, including delay, quota, and load balance. The push server with least delay is preferred to perform commands dissemination. However, some push servers have message quotas limitation. What's more, if only one push sever is used to send commands, botnet may be exposed due to anomaly heavy pushing overhead. Sophisticated load balance technology can improve stealthiness of MUSE.

In order to maintain better utilization of push server, botmaster can establish a dynamic weight vector to record server status. We use a dynamic feedback mechanism to adjust the server weights. By constantly monitoring server load

information, the feedback mechanism can calculate an integrated load balance value. When a server load value is low, the feedback mechanism increases its weight to expand the number of connections. The dynamic feedback mechanism can maintain weight variation within a specific range to achieve good server utilization. Within a fixed period of time  $T$ , two parameters are measured to adjust the weigh vector:

*Input<sub>j</sub>*: it indicates pushing contribution ratio for a push server  $j$ . The value equals the ratio of received push requests  $N_j$  to the average received push requests for all push servers. This parameter is related with the request number from botmaster and is independent of push server itself.

$$Input_j = \frac{N_j \times n}{\sum_{k=1}^n N_k} \quad (4)$$

*Left<sub>j</sub>*: it indicates the remaining push ability for server  $j$ .  $M_j$  means the number of requests sent by a push server  $j$ . It is divided by the maximum quota of a push server and is reset to 0 in the next day. The remaining request quota is related with push server performance. The maximum quota can be learned from push server development documents. In particular, if the maximum number of requests quota is unlimited, then  $Max_j$  can be set to a predefined value. If there is no quota left, then the server is directly set unusable.

$$Left_j = \frac{M_j}{Max_j} \quad (5)$$

With the above two parameters, the feedback mechanism updates the *weight* of server  $j$  every  $T$  time using the Eq. 6.

$$weight_j = weight_j + \alpha \times \sqrt[3]{1 - Input_j - Left_j} \quad (6)$$

$\alpha$  is a feedback parameter. The *weight* value is adjusted in the range of  $[default, \beta \times default]$ . The *default* is initialized by path delay measurement. The path delay can be used as the major factor to evaluate the performance of push server. The  $\beta$  is a range parameter. In practice, when all *weights* are less than the *default*, all servers are idle. On the other hand, if all *weights* exceed  $\beta \times default$ , it means the all servers are overloaded. Botmaster should reduce the number of push requests.

After getting weight vector of push servers, we apply a weight round-robin (WRR) scheduling algorithm to schedule servers for pushing commands. The push server with higher weight will be assigned more push requests than those with lower weight. Different from original WRR algorithm, the proposed scheduling algorithm dynamically adjusts the weight after each round according to Eq. 6.

The improved dynamic WRR algorithm is presented in Fig. 4. Here variable  $i$  indicates index of the current server used for pushing and is initialized with -1.  $cw$  is the current weight in scheduling and is initialized with 0.  $max(weight\_vector)$  is the maximum value in weight vector.  $gcd(weight\_vector)$  is the greatest common divisor of weight vector.  $Input_j, Left_j, N_i, M_i$  are global variables that appeared in Eqs. 4-6.

---

```

Dynamic Weight Round-Robin(weight_vector, last_select, last_cw, push_num)
cw = last_cw ;
i = last_select;
while true do
  i = (i + 1) mod n; // n is the number of push servers
  if i == 0 then
    for each weightj ∈ weight_vector do
      //dynamically update weight in vector
       $Input_j = \frac{N_j \times n}{\sum_{k=1}^n N_k}$ 
       $Left_j = \frac{M_j}{Max_j}$ 
       $weight_j = weight_j + \alpha \times \sqrt[3]{1 - Input_j - Left_j}$ 
      if weightj < default then
        weightj = default
      end if
      if weightj > β × default then
        weightj = β × default
      end if
    end for
    cw = cw - cw_gcd(weight_vector);
    if cw ≤ 0 then
      cw = max(weight_vector);
      if cw == 0 then
        return null;
      end if
    end if
  end if
if weight_vector[i] ≥ cw then
  Ni = Ni + push_num;
  Mi = Mi + push_num;
  return i, cw;
end if
end while

```

---

**Fig. 4.** Dynamic Weight Round-Robin algorithm

## 4.2 Servant Bots Selection

To scale up the botnet, a hybrid architecture is proposed in Sect. 3. Here we discuss how to select servant bots from bot groups to construct intermediate layer. The servant bots construct the intermediate layer and enhance the scalability of MUSE. Servant bots consist of static servant and dynamic servant. Static servant bots may be computers or smartphones firmly controlled by the botmaster. Dynamic servant bots can be selected from each group and changed to avoid being detected.

We adopt the LEACH(Low Energy Adaptive Clustering Hierarchy) protocol [6] to select servant bots. It was designed to lower the energy consumption required to create and maintain clusters for improving the life time of a wireless

sensor network. We apply LEACH to improve the stealthiness because a servant bot should avoid working too long time by dynamically substitution. Each bot in the group uses a stochastic algorithm at each round to determine whether it will become a servant bot in this round.

To select a servant bot, each bot determines a random number between 0 and 1. If the number is less than a threshold  $T(n)$ , the  $bot_n$  becomes a servant bot for the current round. The threshold is set as follows:

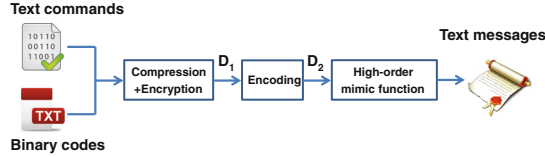
$$T(n) = \frac{P}{1 - P \times (r \bmod \frac{1}{P})} \quad \forall n \in G \quad (7)$$

$$T(n) = 0, \quad \forall n \notin G \quad (8)$$

with  $P$  as the servant bot probability;  $r$  as the number of the current round and  $G$  as the set of bots that have not been servant bots in the last  $1/P$  rounds.  $G$  can be reset over a period of time since we do not require every bot becomes a servant exactly once with  $1/P$  rounds, which is different from the original LEACH.

### 4.3 Message Preparation for Command Transmission

Since Android applications can use dynamic Java class loading to install new functions, a botmaster can send either text commands or binary codes (i.e., dex file) to bots. To evade the detection and fulfill the requirement of push services that only support text messages, MUSE adopts the procedure shown in Fig. 5 to prepare messages for command transmission.



**Fig. 5.** Procedure of message preparation.

To prevent defenders from injecting fake commands, the botmaster generates a pair of public/private keys,  $\langle K_{pr}, K_{pu} \rangle$ , and embeds the public key  $K_{pu}$  in the bot program. After compromising a mobile device, the bot program randomly generates a symmetric encryption key  $K_s$ , encrypts it with  $K_{pu}$ , and sends  $\{K_s\}_{K_{pu}}$ , which denotes the encrypted  $K_s$  using key  $K_{pu}$ , to the botmaster. Note that only the botmaster can obtain  $K_s$  from  $\{K_s\}_{K_{pu}}$  using its private key  $K_{pr}$ . After that,  $K_s$  will be used along with  $\langle K_{pr}, K_{pu} \rangle$  to encrypt/decrypt data transferred between the bot and the botmaster. Moreover, the bot will periodically update  $K_s$  or do it after receiving a certain instruction from the botmaster.

To deliver commands or binary codes through text messages, the botmaster first compresses them, and encrypts them along with a sequence number (i.e.,  $I$ )

and a nonce (i.e.,  $\sigma$ ) using  $K_s$ . The output is concatenated with  $\{\sigma \oplus I \oplus K_s\}_{K_{pr}}$  to generate  $D_1$ . After that, the encoding module adds forward error correction (FEC) code to  $D_1$  and then divides  $D_1$  into blocks. The FEC allows bots to recover the data even if some blocks are lost. Each block is encoded through Base64 in order to represent binary data using ASCII. Moreover, the botmaster uses high-order mimic functions [21] to transform each block in  $D_2$  into English text with similar statistical properties in order to evade the detection. Finally, the botmaster sends these English text messages to bots through push services. After receiving the text message, a bot first recovers  $D_2$  using the corresponding inverse function and then extracts the binary data through base64 and FEC decoding. Using the embedded public key  $K_{pu}$  and  $K_s$ , the bot can extract the original data, authenticate it and verify its integrity. A bot employs a similar approach to send collected data to the botmaster.

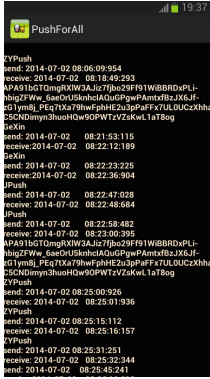
## 5 Implementation

We have implemented MUSE by integrating ten push services including GCM, JPush, XGPush, ZYPush, GeXinPush, Airbop, Parse, Pushwoosh, Pushbots and PushApps. Since the last five push services rely on GCM services, we only use the first six push services in the experiments. The push services' properties are listed in Table 1. Most of popular push services are free and some of them also provide premium services to paid users. All push services in the Table 1 support statistical reports, including online users, amount of messages sent, and amount of messages read. Moreover, we recruited 15 volunteers and installed MUSE bots into their Android phones.

**Table 1.** Properties of push services

Push services	Registration parameters	Frequency limit	Size limit	Connection server
GCM	SenderId	token bucket scheme	4 KB	HTTP POST, CCS
Airbop	SenderId, Appkey, Appsecret	N/A	4 KB	HTTP POST, CCS
Parse	ApplicationID, ClientKey	N/A	N/A	HTTP PUT, POST
PushWoosh	ApplicationCode, GCMSenderID	N/A	N/A	HTTP POST
PushBots	ApplicationID, GCMSenderID	N/A	N/A	HTTP POST
PushApps	AppToken, GCMSenderID	N/A	N/A	HTTP POST
JPush	Appkey	<600/min	1 KB	HTTP POST
GexinPush	AppID, AppSecret, Appkey	N/A	2 KB	HTTP OpenService
XinGePush	AccessID, AccessKey	<20/min for broadcast	4 KB	HTTP Post
ZYPush	Appkey	N/A	1 KB	HTTP Post

Thus, we construct a real-life, small scale MUSE botnet with 15 bots. The Android application of MUSE bot can receive messages from different push servers as shown in the Fig. 6a. We then experiment four different attack commands: **contactinfo**, **gpsinfo**, **pic-upload** and **applist**. These commands require bots to upload contact list, GPS location information, photographs and applications list to a specified URL.



(a) Push contents

com.android.soundrecorder	null	null
com.android.voicedialer	null	null
com.android.defcontainer	null	null
com.android.launcher	com.android.launcher2.Laun	
com.android.quicksearchbox	com.android.quickse	
com.android.contacts	com.android.contacts.Conta	
com.android.inputmethod.latin	null	null
com.android.phone	com.android.phone.PhoneApp	
com.android.calculator2	null	null
com.android.proxyhandler	null	null
com.android.htmlviewer	null	null
com.android.providers.calendar	null	null
com.android.bluetooth	com.android.bluetooth.btse	
com.android.inputdevices	null	null
com.android.wallpaper.holospiral	null	nu
com.android.calendar	com.android.calendar.Calen	

(b) Applications list

**Fig. 6.** Attack demonstration

## 5.1 Incremental Update

Once some accounts are blocked by defenders, botmaster prefers to update the push service accounts in bots. They can apply for new accounts to continue pushing services. As we analyzed in Sect. 3.4, botmaster can disseminate commands, including update information, to bots even if there is only one available account left. A simple update method is to recompile the latest version of the bot program with new accounts. Then botmaster dispatches the latest version to bots. This method will increase the network traffic dramatically, which can be easily detected by anomaly detection methods. Therefore, it requires a light-weight update method with low traffic consumption. We apply an incremental update method, which uses patch files to update bots. These accounts information can be compiled as a patch file with limited size. The patch file can implement incremental update using technology like Smart App update provided by Google.

The principle of incremental update is simple. Developers compare the old version with the new version and get the difference. Then they make update patch file and publish it to users. After downloading the patch, users need to integrate the old version and the update patch to generate the new version. In addition, when developers make the patch, the latest version should be compared with each previous version because users may use different old versions.

Botmaster can force all bots to update to the latest version, which can decrease the maintenance costs for servers and clients. The bots can use two methods to configure push service accounts. One is to put the encrypted accounts in the class files and the other one is to store them in the manifest file. The latter is more efficient than the former, because the botmaster can perform efficient incremental update by sending a patch file that only changes the manifest file. Note that some push services do not support the latter update mechanism, such as Parse, Pushbots.

**Table 2.** Incremental Update

Push services	Update file	App size	Patch size
Pushwoosh	AndroidManifest.xml	362 KB	4 KB
Parse	PushActivity.java	581 KB	500–540 KB
PushBots	PushActivity.java	322 KB	280–300 KB
PushApps	PushActivity.java	331 KB	280–300 KB
ZYPush	AndroidManifest.xml	416 KB	4 KB
XGPush, GeXinPush, JPush(Integrated)	AndroidManifest.xml	1407 KB	10 KB

The size of patch file is compared with that of the original application file in the Table 2. It shows that incremental update with manifest file can dramatically decrease the update cost. But if the upgrade patches use the java file (i.e., bytecode generated from java files), the patch sizes are very large, which are almost as same as the original files.

## 5.2 Bot Communications

Most of push services support push API for app developers. For example, some push servers provide REST API (Representational State Transfer), which defines a set of push services as web services. Application servers or smartphones can use HTTP methods explicitly to perform pushing operations. It means a smartphone can not only receive messages, but also send pushing messages with REST support. It provides a mechanism for bots to send messages to botmaster or other bots through push services. Current push-styled botnets [10, 23] only support one-way communications, which is from botmaster to bots. In MUSE, bots can use bi-direction communications to send feedback reports to botmaster. This also enables bots to act as servants to construct botnet intermediate layer.

In the hybrid structure shown in Fig. 2, there are three types of nodes: C&C server, servant bot and client bot. We can establish a communication channel for each two of them and propose three channels according to Botnet Triple-Channel Model (BTM) [3]. In MUSE, there are three types of channels:

1. One-way channel from C&C server to client bot. Like the conventional message push mobile botnet [10, 23], C&C server directly uses push services to disseminate commands to client bots with this one-way channel.
2. Bi-direction channel between servant bot and client bot. We find that the SDKs provided by most push services are designed for PC instead of smartphone. Fortunately, they also provide HTTP Restful APIs, and hence mobile applications can send HTTP requests to realize the push function. This functionality enables all mobile bots to establish bi-direction communication channel. In other words, client bots can not only receive commands but also submit push requests.
3. One-way channel from bots to C&C server. Since there is no direct communication channel from bots to C&C server, one possible approach is to use

third-party servers, such as email or network disk [12], to upload data collected on victims. This channel can also utilize a short URL or URL-Flux and other existing technologies to enhance the robustness and stealthiness. After bots upload data to file servers, C&C server then downloads data from them.

In BTM, traditional C&C channel was substituted by three independent sub-channels, denoting as Command Download Channel (CDC), Registration Channel (RC) and Data Upload Channel (DUC), respectively. CDC is responsible for commands distribution. RC is responsible for collecting fundamental registration information. DUC is responsible for collecting and storing the uploaded information. In MUSE, BTM can be completely enhanced with our proposed bi-direction channels. In particular, push servers are used to construct CDC. File servers can help bots upload registration information as RC and steal sensitive information on victims as DUC. The three channels are listed and compared with BTM in Table 3.

**Table 3.** Triple channel model

Channel	BM $\leftrightarrow$ SB <sup>a</sup>	BM $\leftrightarrow$ CB <sup>b</sup>	SB $\leftrightarrow$ CB <sup>c</sup>
Registration channel	File servers		
Command download channel	Push servers		
Data upload channel	File servers		File servers

<sup>a</sup>Between botmaster and servant bot.

<sup>b</sup>Between botmaster and client bot.

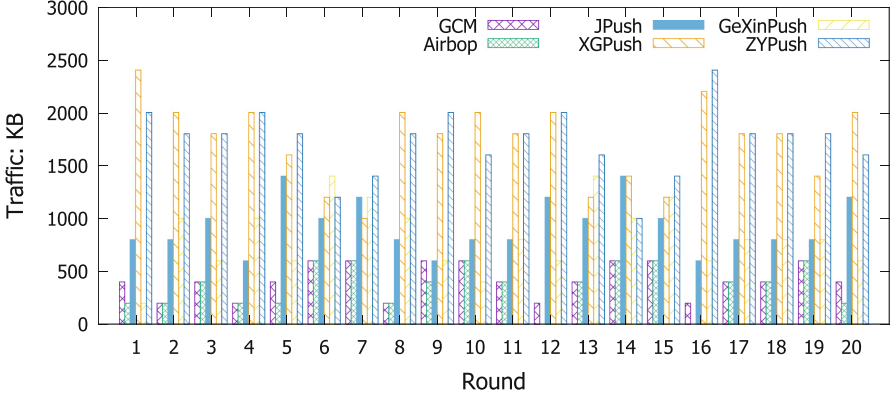
<sup>c</sup>Between servant bot and client bot.

## 6 Evaluation

### 6.1 Stealthiness

The stealthiness is critical for the botnet to circumvent the detection. We first use simulation to evaluate the stealthiness of MUSE. In this experiment, six push servers are used to construct MUSE C&C and 600 bots are divided into 3 groups. Botmaster will send 200 commands to each bot. There are totally 120,000 push messages sent from the push servers. The size of each push message containing commands is assumed to be 1 KB. When perform pushing operation, the botmaster selects one push server for each bot group according to the proposed dynamic WRR algorithm. The weight vector for six push servers is initialized with path delay measurement results. In each round, the botmaster disseminates 10 consecutive commands via six push servers and the traffic volume of each server is recorded in Fig. 7. By observing the traffic volume of six push servers in each round, we find that push requests disperse among six servers according to their weight. It is worth noting that since the pushed botnet traffic varies in different rounds, it is not easy for the defender to find botnet traffic patterns.





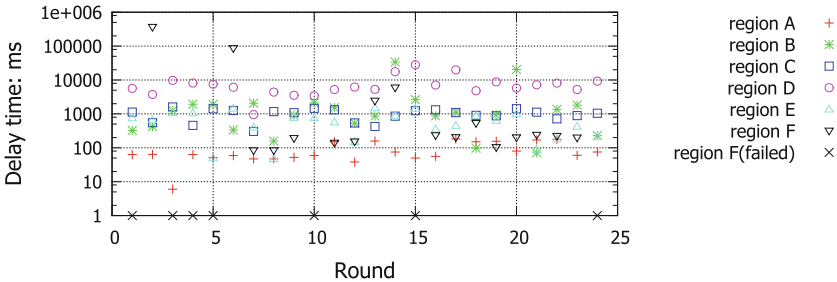
**Fig. 7.** Botnet traffic disperse among different push servers (Color figure online).

At the same time, the dynamic WRR algorithm can achieve good load balancing. Moreover, MUSE can make full use of all push services according to their performance, thus improving the scalability of the botnet.

## 6.2 Controllability

The controllability means whether the bot can receive commands efficiently from the botmaster. The smaller delay of disseminations will lead to the better controllability of botnet. Controllability is important for the botnet, especially during period of performing some synchronized attacks.

We first measure the delay of GCM-based mono-push-server botnet for a whole day in 6 different regions. The measurement is performed every hour and there are total 24 rounds for one day. The delay specifies how long it takes for a push command to travel across the network from botmaster to a bot. We repeat each experiment 5 times and take statistical measures.



**Fig. 8.** Command delay for mono-push-server botnet (Color figure online).

Figure 8 shows the performance of mono-push-server botnet is not stable in some regions. Especially in region F (Nanjing, China), some messages delay is longer than 10s and even 7 messages are lost (note the logarithmic scale on the y-axis). The poor pushing performance is fatal for a mono-push-server botnet because it means the commands can not be reliably sent to bots.

Then we evaluate delay of command dissemination for MUSE. The experiment is conducted in region F where GCM performance is not stable. We compare the dynamic weight round-robin algorithm, which is implemented in MUSE, with the other two scheduling algorithms: round-robin scheduler and random scheduler. The experiments are conducted once per hour for a whole day and 24 rounds results are recorded. In each round the botmaster sends 10 commands with three different scheduling algorithms. As illustrated in the Fig. 9, the performance of the weight round-robin is superior to the other two. In general, MUSE with weigh round-robin algorithm can disseminate commands efficiently and outperform the mono-push-server botnet in stability and controllability.

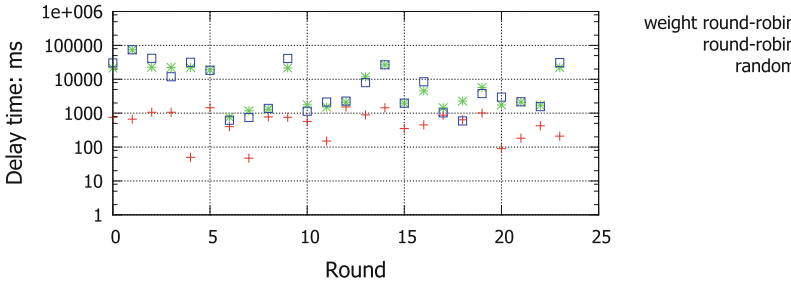


Fig. 9. Command delay for MUSE (Color figure online).

## 7 Conclusion

We propose MUSE to construct mobile botnets using multiple message push servers. It can avoid the single point of failure problem in mono-push-server structure and efficiently improve the robustness. MUSE has new architectures and group management to improve the scalability and controllability of botnet. Moreover, to keep stealthy, we design new algorithms for botmaster to select push servers and servant bots. We have implemented MUSE and the experimental results demonstrate MUSE's capability. In future work, we will investigate how to defend against such mobile botnets.

**Acknowledgement.** This work was supported in part by the Hong Kong GRF/ECS (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396, 61202353, 61272084), the HKPolyU Research Grant (G-YBJX).

## References

1. Anagnostopoulos, M., Kambourakis, G., Gritzalis, S.: New facets of mobile botnet: architecture and evaluation. *Int. J. Inf. Secur.* 1–19 (2015)
2. Chen, W., Yin, C., Zhou, S., Yan, X.: Cloud-based mobile botnets using multiple push servers. In: 2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), pp. 183–189. IEEE (2015)
3. Cui, X., Fang, B., Liao, P., Liu, C.: Advanced triple-channel botnets: model and implementation. In: *Proceedings of CCS* (2012)
4. Cui, X., Fang, B., Yin, L., Liu, X., Zang, T.: Andbot: towards advanced mobile botnets. In: *Proceedings of LEET* (2011)
5. Eslahi, M., Rostami, M.R., Hashim, H., Tahir, N.M., Naseri, M.V.: A data collection approach for mobile botnet analysis and detection. In: 2014 IEEE Symposium on Wireless Technology and Applications (ISWTA), pp. 199–204, September 2014
6. Handy, M., Haase, M., Timmermann, D.: Low energy adaptive clustering hierarchy with deterministic cluster-head selection. In: *Proceedings of IEEE MWCN* (2002)
7. Hua, J., Sakurai, K.: Botnet command and control based on short message service and human mobility. *Comput. Netw.* **57**(2), 579–597 (2013)
8. Karim, A., Shah, S.A.A., Salleh, R.: New perspectives in information systems and technologies. In: Rocha, Á., Correia, A.M., Tan, F.B., Stroetmann, K.A. (eds.) *Mobile Botnet Attacks: A Thematic Taxonomy*, vol. 2, pp. 153–164. Springer International Publishing, Cham (2014). ISBN=978-3-319-05948-8
9. Khattak, S., Ramay, N., Khan, K., Syed, A., Khayam, S.: A taxonomy of botnet behavior, detection, and defense. *IEEE Commun. Surv. Tutor.* **16**(2), 898–924 (2014)
10. Lee, H., Kang, T., Lee, S., Kim, J., Kim, Y.: Punobot: mobile botnet using push notification service in android. In: Kim, Y., Lee, H., Perrig, A. (eds.) *WISA 2013*. LNCS, vol. 8267, pp. 124–137. Springer, Heidelberg (2014)
11. Luo, X., Chan, E., Zhou, P., Chang, R.: Robust network covert communications based on TCP and enumerative combinatorics. *IEEE Trans. Dependable Secure Comput.* **9**(6), 890–902 (2012)
12. Luo, X., Zhou, H., Yu, L., Xue, L., Xie, Y.: Characterizing mobile \*-box applications. *Comput. Netw.* **103**, 228–239 (2016)
13. Mullaney, C.: Android.Bmaster: a million-dollar mobile botnet (2012). <http://goo.gl/sxpoNN>
14. Mulliner, C., Seifert, J.P.: Rise of the iBots: owning a telco network. In: *Proceedings of IEEE MALWARE* (2010)
15. Pieterse, H., Olivier, M.: Design of a hybrid command and control mobile botnet. In: *Proceedings of the 8th International Conference on Information Warfare and Security, ICIW 2013*, p. 183. Academic Conferences Limited (2013)
16. Rodríguez-Gómez, R., Maciá-Fernández, G., García-Teodoro, P.: Survey and taxonomy of botnet research through life-cycle. *ACM Comput. Surv.* **45**(4), 45 (2013)
17. Silva, S., Silva, R., Pinto, R., Salles, R.: Botnets: a survey. *Comput. Netw.* **57**(2), 378–403 (2013)
18. Singh, K., Sangal, S., Jain, N., Traynor, P., Lee, W.: Evaluating bluetooth as a medium for botnet command and control. In: Kreibich, C., Jahnke, M. (eds.) *DIMVA 2010*. LNCS, vol. 6201, pp. 61–80. Springer, Heidelberg (2010)
19. Traynor, P., Lin, M., Ongtang, M., Rao, V., Jaeger, T., McDaniel, P., La Porta, T.: On cellular botnets: measuring the impact of malicious devices on a cellular network core. In: *Proceedings of ACM CCS* (2009)

20. Wang, P., Sparks, S., Zou, C.C.: An advanced hybrid peer-to-peer botnet. *IEEE TDSC* **7**(2), 113 (2010)
21. Wu, Z., Gianvecchio, S., Xie, M., Wang, H.: Mimimorphism: a new approach to binary code obfuscation. In: *Proceedings of ACM CCS* (2010)
22. Zeng, Y., Shin, K.G., Hu, X.: Design of SMS commanded-and-controlled and P2P-structured mobile botnets. In: *Proceedings of WiSec* (2012)
23. Zhao, S., Lee, P., Lui, J., Guan, X., Ma, X., Tao, J.: Cloud-based push-styled mobile botnets: a case study of exploiting the cloud to device messaging service. In: *Proceedings of ACSAC* (2012)
24. Zhou, Y., Jiang, X.: An analysis of the AnserverBot trojan (2011). <http://goo.gl/Dz8qda>
25. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: *Proceedings of IEEE Symposium on Security and Privacy* (2012)

Information Security and Privacy

21st Australasian Conference, ACISP 2016, Melbourne,  
VIC, Australia, July 4-6, 2016, Proceedings, Part I

Liu, J.K.; Steinfeld, R. (Eds.)

2016, XVIII, 543 p. 114 illus., Softcover

ISBN: 978-3-319-40252-9