Hindawi
Security and Communication Networks
Volume 2018, Article ID 8510256, 12 pages
https://doi.org/10.1155/2018/8510256

*Research Article*

# Design and Analysis of Push Notification-Based Malware on Android

**Sangwon Hyun**,[1] **Junsung Cho**,[2] **Geumhwan Cho**,[2] **and Hyoungshick Kim**[2]

[1]*Department of Computer Engineering, Chosun University, Gwangju, Republic of Korea*
[2]*Department of Software, Sungkyunkwan University, Suwon, Republic of Korea*

Correspondence should be addressed to Hyoungshick Kim; hyoung@skku.edu

Received 14 March 2018; Accepted 6 June 2018; Published 9 July 2018

Academic Editor: Francesco Palmieri



Establishing secret command and control (C&C) channels from attackers is important in malware design. This paper presents design and analysis of malware architecture exploiting push notification services as C&C channels. The key feature of the push notification-based malware design is *remote triggering*, which allows attackers to trigger and execute their malware by push notifications. The use of push notification services as covert channels makes it difficult to distinguish this type of malware from other normal applications also using the same services. We implemented a backdoor prototype on Android devices as a proof-of-concept of the push notification-based malware and evaluated its stealthiness and feasibility. Our malware implementation effectively evaded the existing malware analysis tools such as 55 antimalware scanners from VirusTotal and SandDroid. In addition, our backdoor implementation successfully cracked about 98% of all the tested unlock secrets (either PINs or unlock patterns) in 5 seconds with only a fraction (less than 0.01%) of the total power consumption of the device. Finally, we proposed several defense strategies to mitigate push notification-based malware by carefully analyzing its attack process. Our defense strategies include filtering subscription requests for push notifications from suspicious applications, providing centralized management and access control of registration tokens of applications, detecting malicious push messages by analyzing message contents and characteristic patterns demonstrated by malicious push messages, and detecting malware by analyzing the behaviors of applications after receiving push messages.

## 1. Introduction

*Stealthiness* is a key requirement of malware for the persistence of attack. *Remote triggering* can enhance the stealthiness of malware by allowing attackers to periodically trigger and execute their malware only whenever they want, while normally hiding the presence of malware from victims. In addition, it is also desirable for attackers to send attack commands to their malware for the flexible control of the malware's behaviors. Some examples of attack commands include the following: "perform a DDoS attack against foo.com at a certain transmission rate during a certain period of time"; "collect the screen lock password of the device"; and "gather pictures taken or SMS messages received on a certain date." Such features as remote triggering and delivering attack commands essentially require communication channels between attackers and their malware on victims' devices, and these communication channels are commonly referred to as *command and control (C&C) channels*.

Remote C&C channels of malware should meet the following requirement for their stealthiness. Many malware scanner tools can try to intercept the network traffic of the tested application and analyze the captured traffic for detecting suspicious behavior of malware. Thus, a secret C&C channel of malware should be established to avoid detection by scanners using the network traffic analysis.

Several groups of researchers presented mobile botnets exploiting push notification services to establish their C&C channels [1–3] and evaluated the feasibility of push notification services as C&C channels in terms of stealthiness, scalability, and efficiency. However, those previous studies mainly focused on the design of botnet using push notification services. In addition, those studies offered only a brief discussion on how to mitigate push notification-based mobile botnets.

This paper is based on our preliminary work [4]. In this paper, we extend the backdoor implementation presented in the preliminary work [4] to develop a more *generic* and *flexible* push-based backdoor design that can be applied to any type of malware (e.g., botnet, backdoor, and other malware types) for the purpose of setting up a C&C channel from the attacker via push notifications. In addition, we comprehensively analyze the attack procedure of push notification-based malware and propose a set of defense mechanisms to mitigate push-based malware in each step of the attack procedure.

Push notification services can be used to secretly hide the communication channel between malware and remote network parties (e.g., C&C server). This is because push notification services are also popularly used for benign applications, and thus this makes detection difficult. Moreover, push notification-based malware requires permissions that are only needed to use push notification services; thus our malware is resilient against malware detection techniques using permissions, which are also very commonly used in antimalware scanners. Because the behaviors of malware can easily be switched on and off by push messages from the attacker, the chance of detection can be reduced.

Sending a push message to malware requires the attacker's prior knowledge of the registration token of the installed malware. Therefore, a covert way of delivering the registration token of the malware to the attacker is needed for push notification-based malware. In this paper, we suggest using a public cloud-based database service (e.g., Firebase Realtime Database) as a medium of registration token delivery. Incorporating a public intermediate service makes it hard to distinguish the malicious activity of delivering the registration token from other benign activities using the same service.

To show the feasibility of our malware design, we implemented backdoor (as a representative type of push notification-based malware) on Android 5.1 and evaluated its stealthiness and feasibility. The evaluation results showed that existing malware detection tools based on static and/or dynamic analysis were ineffective to detect our backdoor implementation. Specifically, all the tested antimalware scanners (VirusTotal [5] and SandDroid [6]) failed to identify our backdoor as malware. In addition, we observed that our backdoor successfully cracked about 98% of all the tested unlock secrets (either 4-digit PINs or patterns) in 5 seconds with only a fraction (less than 0.01%) of the total power consumption of the device.

To reduce the risk of such malware, we carefully analyzed the push notification-based malware architecture and the attack procedure over it and consequently suggest several possible defense strategies to mitigate push notification-based malware. Our defense strategies include filtering subscription requests for push notifications from suspicious applications, providing centralized management and access control of registration tokens of applications, detecting malicious push messages by analyzing message contents, and detecting malware by analyzing the behaviors of applications after receiving push messages.

The remainder of this paper is organized as follows. The next section overviews the related work, and Section 3 gives a brief overview of Firebase. Section 4 defines the threat model considered in this paper. Section 5 presents the design of push notification-based malware. Section 6 describes the evaluation of our proof-of-concept malware implementation. Section 7 discusses possible defense strategies to mitigate push notification-based malware. Section 8 concludes this paper.

## 2. Related Work

There exist a number of malware examples that exploit various communication methods available on mobile devices. Several malware misused Short Message Service (SMS) for their malicious activities. Zeng et al. [7] presented a mobile botnet that uses SMS as a medium to deliver bot commands and studied a P2P network architecture suitable for the botnet in terms of the number of message transmissions and delay. Hua et al. [8] analyzed the impact of network topology on dissemination cost of bot commands over SMS-based mobile botnets. Mulliner et al. [9] and Anagnostopoulos et al. [10] proposed mobile botnets that use a hybrid of SMS, HTTP, and/or Bluetooth as a C&C medium. There also exists an SMS-based Trojan [11] that causes financial loss to users by sending messages to premium rate numbers without the users' recognition.

There is another group of malware examples which is based on other communication methods such as Bluetooth and social networking services. Android.Obad.OS [12] is a worm that propagates copies of itself via Bluetooth. Singh et al. [13] studied the feasibility of Bluetooth as C&C channels of mobile botnets. DR-SNBot [14] and Andbot [15] are mobile botnets that use social networking services as C&C channels. In these botnets, the botmaster posts bot commands disguised as image files on blog sites; then bots download the image files and extract the bot commands from the image file.

Most relevant to our research, several groups of researchers presented mobile botnets exploiting push notification services as their C&C channels [1–3] and evaluated the feasibility of push notification services as C&C channels in terms of stealthiness, scalability, and efficiency. In particular, Zhao et al. [1] proposed an approach that exploits upstream push messages for each bot to secretly deliver its registration token to the botmaster. Chen et al. [3] suggested the use of multiple push servers, not relying on a single server, to provide improved scalability and fault tolerance of mobile botnets. In our preliminary work [4], we presented a design of Android backdoor exploiting push notification services. Based on those previous studies, we designed a generic malware architecture exploiting push notification services to support remote triggering and flexible control of malware and implemented a backdoor prototype on Android devices to show the feasibility of that malware architecture. We also suggested possible defense strategies to mitigate push notification-based malware.

## 3. Overview of Firebase

Firebase (https://firebase.google.com) is a mobile application development platform to facilitate the development of applications using cloud-based services such as Firebase

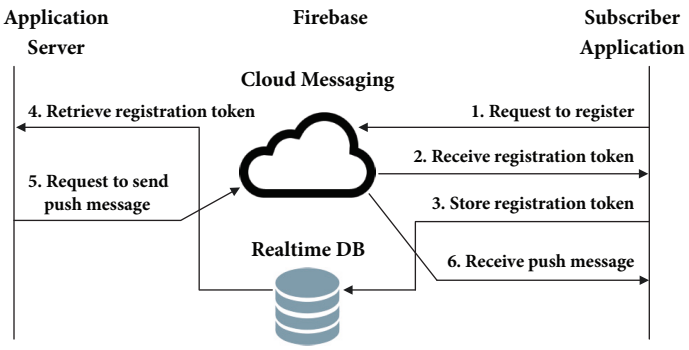


Figure 1: Overview of Firebase.

Cloud Messaging (FCM, https://firebase.google.com/products/cloud-messaging/) and Firebase Realtime Database (FRD, https://firebase.google.com/products/realtime-database/).

FCM is a cloud-based messaging service that allows third-party developers to send push messages to their applications on user devices. Figure 1 shows the infrastructure and the overall procedure of a push messaging service based on FCM. In this infrastructure, the application server managed by a third-party developer sends messages to the subscriber application on a user device via the FCM server.

To develop an application that can receive push messages via FCM, the developer first needs to sign up for a developer account of his (or her) application in Firebase with his (or her) Google ID and password and the package name of the application. As a result, the developer is given an application ID (a unique identifier of the developer's application), a sender ID (a unique identifier assigned to the developer as a push message sender), and an authorization key (a credential required to request the FCM server to send push notifications) from the FCM server. The developer then preconfigures his (or her) application package with the application ID and the sender ID given.

After the application package has been installed on a user device, the application first registers itself with the FCM server (Step 1 in Figure 1). This registration request contains the device ID (a unique identifier of the user device hosting the application) as well as the application ID and the sender ID that have been preconfigured with the application package by the developer. Once receiving this registration request, the FCM server generates and delivers to the application a *registration token* that uniquely identifies the application on this particular user device (Step 2).

Sending a push message to a subscriber application requires prior knowledge of the registration token of the recipient application. To achieve this, we use FRD, which allows sharing information between a subscriber application and an application server. After receiving the registration token from the FCM server, the subscriber application stores it in the FRD (Step 3), and then the application server downloads the registration token from the FRD (Step 4).

The application server is now ready to send a push message to the subscriber application. To send a push message, the application server sends the message content, the registration token of the target application, and the authorization key of the developer to the FCM server (Step 5). If the authorization key and the registration token are valid, the FCM server finally sends the message to the subscriber application (Step 6).

## 4. Threat Model

This section describes the threat model considered in our malware design.

To effectively distribute push notification-based malware, an attacker embeds his (or her) malicious codes into the codes of a normal application to disguise his (or her) malware as a legitimate application and distributes this repackaged application to victims. This method is referred to as *repackaging* and is known as a popular method to distribute mobile malware [16]. In practice, this repackaging method increases the chance that a victim installs the malware without being aware of the risk because the malware is embedded secretly into a legitimate application. Furthermore, the repackaged malware can be delivered to a victim via social engineering methods [17] (e.g., sending a gift app), and this further increases the chance that the malware is installed on the victim's device. If an insider attacker (e.g., friends, colleagues, or family members) often gains physical access to a victim's device, it is also possible that the attacker installs the malware secretly on the victim's device. Thus we assume that the malware is installed secretly on the smartphone of a victim.

For application repackaging, the attacker first selects a popular application as a host application for embedding the malicious codes and downloads the Android application package (APK) file of the selected application. The attacker then reconstructs the original codes of the host application in human readable format (e.g., smali codes) by unarchiving and decompiling the APK file of the host application. The next step is to add some malicious codes to the original application codes by either modifying the existing code file or adding a new separate code file and also modifying `AndroidManifest.xml` for additional configurations required for the malicious activities. The attacker then generates a new APK file by rebuilding the modified

application codes (with the malicious codes), signs the new APK file using his (or her) private key, and attaches the self-signed certificate of his (or her) public key. It is worth noting that the use of the self-signed certificate allows the attacker to claim whatever identity he (or she) wants in the certificate. The attacker distributes the new APK file attached with the certificate to mobile application stores. This repackaging method increases the chance that a victim installs the malware without being aware of the risk because the malware is embedded secretly into a legitimate application. We assume that the repackaged malware is installed secretly on the smartphone of a victim.

In this paper, we consider two types of malware, *backdoor* and *DDoS (Distributed Denial-of-Service) bot*, as examples of malware. The following describes our assumptions on each type of malware.

*4.1. Backdoor.* In this attack scenario, we assume an *insider* attacker (e.g., friends, colleagues, or family members) who often gains physical access to a victim's smartphone. A previous study [18] demonstrated that smartphone users are mostly concerned about preventing unauthorized access to the private information on their devices from such insiders. We assume that the victim's smartphone is protected with a screen lock mechanism that can provide a proper level of security (e.g., 4-digit PIN or screen lock pattern). We also assume that the victim regularly changes the unlock secret of the smartphone. Under these conditions, the objective of the *backdoor* installed on the victim's smartphone is to find the unlock secret of the device and delivers the secret to the attacker without being detected by the victim. That is, the attacker who wants to secretly look into the private information of the victim obtains the unlock secret of the victim's smartphone by triggering the backdoor secretly, unlocks the victim's smartphone, and investigates the private information inside the smartphone. In this way, the attacker can persistently obtain the private information on the victim's smartphone even when the unlock secret is often changed.

*4.2. DDoS Bot.* In this attack scenario, the attacker refers to the botmaster who has the control of bot applications installed on victims' devices. We assume that the attacker already knows the registration tokens of the bot applications so that the attacker can send push messages to the bot applications. FCM supports the maximum payload size of 4 KB (https://firebase.google.com/docs/cloud-messaging/concept-options), and we assume that this payload size is sufficient to contain a triggering signal and some basic information (e.g., the network identifiers of attack target hosts) that is necessary for attacks. The victims' smartphones should have capabilities to perform DDoS attacks. In fact, recent smartphones provide not only powerful computing capabilities but also a persistent Internet connection via cellular networks or Wi-Fi. Also, they support a high data transmission rate via, for instance, LTE (https://www.tomsguide.com/us/best-mobile-network,review-2942.html). Thus smartphones are powerful enough to perform DDoS attacks.

## 5. Design and Implementation of Push Notification-Based Malware

This section describes the design and implementation of push notification-based malware on Android devices. Our implementation is composed of three components: *triggering application*, *Firebase*, and *malware*.

Remote triggering of malware is important for hiding the malware from antimalware tools. We use FCM (Firebase Cloud Messaging) to achieve this by allowing an attacker to trigger malware via push notifications. Because FCM is also used by many normal applications demanding push notification services, it might be difficult for existing antimalware tools to distinguish the traffic of malware from the traffic of benign applications via FCM.

The triggering application is executed on the attacker's device, while malware is executed on the victim's device. In our implementation, Nexus 5X on Android 6.0 and Nexus 5 on Android 5.1 were, respectively, used for the attacker's and the victim's devices. Our malware implementation can be adapted to other Android versions as well. We confirmed that our malware also performs well on lower Android versions than Android 5.1 by slightly modifying the code. Consequently, our malware can currently be applied to 42.3% of all Android smartphones according to the latest statistics on Android version distribution (https://developer.android.com/about/dashboards).

*5.1. Attack Procedure.* This section describes the overall attack procedure in our malware design based on push notification services. As discussed in Section 4, we assume that malware is already installed on a victim's smartphone. The triggering application and malware in our implementation correspond to the application server and the subscriber application in Figure 1, respectively.

After being installed on the victim's smartphone, the malware sends the FCM server a request to register itself as a push message recipient (Step 1 in Figure 1), and the FCM server issues a unique registration token to the malware in response (Step 2). The malware then uploads the received registration token to the FRD (Firebase Realtime Database) (Step 3) so that the triggering application (on the attacker's device) can retrieve the registration token from the FRD (Step 4). After the triggering application obtains the registration token of the malware, the attacker is now ready to send a push message to the malware. Whenever the attacker wants to trigger the malware on the victim's smartphone, the attacker uses his (or her) triggering application to send the FCM server a push notification request that contains the registration token of the malware and a push message to deliver to the malware (Step 5); the push message in the request may contain a signaling message to trigger the malware and an attack command to deliver. Once receiving the push notification request with the registration token, the FCM server checks which application on which device has been assigned this token, which is the malware in this case, and then forwards the push message to the malware (Step 6). The push message that arrives at the victim's device is first delivered to the client-side agent of push notification service, which is running on the victim's device;

```
Authorization : key =" AIzaSyBCkAgv4−iucpSqdpd6g . . . "
{
        message : {
                token : " bk3RNwTe3H0 : CI2k_Hl3pja99tls . . . " ,
                data : {
                        body : <command to trigger and
                                control malware >,
                        . . .
                }
        }
}
```

Figure 2: JSON-formatted push message from triggering application.

then the agent process hands the push message over to the malware. Then the malware is triggered automatically, checks the attack command contained in the received message, and launches the malicious activities specified in the attack command such as cracking the victim's unlock password, accessing some private information of the victim, and transmitting attack packets to the target host of a DDoS attack.

*5.2. Triggering Application.* The triggering application is installed and executed on the attacker's device and requests the FCM server to send push messages to trigger and/or command the malware through HTTP. As already mentioned, the triggering application corresponds to the application server in Figure 1. In fact, any programming language and platform can be utilized to implement and run the triggering application. Figure 2 shows an example push message sent by the triggering application in JSON format. As mentioned in Section 3, `Authorization key` is required to request the FCM server to send push messages. The `token` field encloses the registration token that identifies the target malware to which the message should be eventually delivered.

*5.3. Malware.* This section describes the design and implementation of the malware, which corresponds to the subscriber application in Figure 1. *Stealthiness* is a key property required by the malware. To this end, we carefully designed the malware to provide the following properties. First, by adopting the remote triggering feature, our malware does not have to be always run on a victim's device. Instead, our malware is triggered and executed only when the attacker wants, and this can significantly reduce the chance of being detected by the victim. Second, all communications between the remote triggering application and malware can always be done through a push messaging service that is also popularly used by other benign applications. Hence, it is hard for malware detection tools based on network traffic monitoring to detect our malware. Third, our malware implementation requires the permissions (`WAKE_LOCK`, `ACCESS_NETWORK_STATE`, `INTERNET`, `<package-name>.permission.C2D_MESSAGE`, `com.google.android.c2dm.permission.RECEIVE`) that are only needed to use FCM, and these permissions are also required by other benign applications using FCM. Thus it is also difficult even for security-conscious users to notice the existence of our malware on their smartphones. For the same reason, malware detection techniques using permissions (e.g., [19]) are also ineffective for detecting our malware.

The following subsections describe two types of malware: *backdoor* and *DDoS bot*. In particular, we implemented the backdoor application to validate the feasibility of push notification-based malware.

*5.3.1. Backdoor.* The goal of backdoor application installed on a victim's device is to identify the unlock password of the victim and secretly deliver it to the attacker. Figure 3 shows the overall attack procedure of push notification-based backdoor. The attacker who wants to access the victim's device requests an FCM server to send a push message to the backdoor application that has been installed on the victim's device by using the triggering application (Step 1). The FCM server then sends the requested push message to the backdoor application (Step 2). Once the push message arrives at the victim's device (Step 3), the backdoor application is triggered automatically (Step 4) and identifies the lock mechanism (e.g., PIN or screen lock pattern) being used by the victim (Step 5). The backdoor application is able to know the lock mechanism in use by accessing `locksettings.db` file stored in `/data/system/`. Specifically, `lockscreen.password_type` field in the file indicates which lock mechanism is currently in use with an integer value; the value for PIN is `131072` or `196608`, and the value for screen lock pattern is `65536`.

After identifying the lock mechanism in use, the backdoor application tries to find out the password by executing the cracking module corresponding to the identified lock mechanism (Step 6). When storing a user input password in a file, Android stores the hash value of the password rather than the password itself. Thus cracking the password requires accessing the file where the hash value of the password is stored, and accessing this special file requires a root privilege. For this reason, it is assumed that the victim's device is rooted before the backdoor application is installed. While this assumption may seem rather strong, it is one that appears to be commonly made in practice. There exist several survey results showing that a large number of Android devices were rooted by device owners for the purpose of getting rid of unnecessary built-in apps or updating to the latest version of Android. According to the official report from Google in 2016 [20], about 5.6% of all Android users were using rooted devices, either intentionally or due to security bugs. Furthermore, the survey results in [21] showed that about 7% of all respondents were using rooted Android devices. Tencent also conducted a survey of Android users in China during 2014 [22] and found that about 80% of the survey participants were using rooted Android devices. To make matters worse, security bugs can make Android devices rooted forcibly [23].

We implemented the password cracking algorithm shown in Algorithm 1 to efficiently guess a victim's unlock password from the hashed password stored in the victim's device. In this algorithm, first of all, the backdoor application reads the hashed password (in line (1)) from either `gesture.key` (for screen lock pattern) or `password.key` (for PIN). The backdoor application also retrieves the corresponding salt (in line (7)) from `locksettings.db`, especially for PIN.

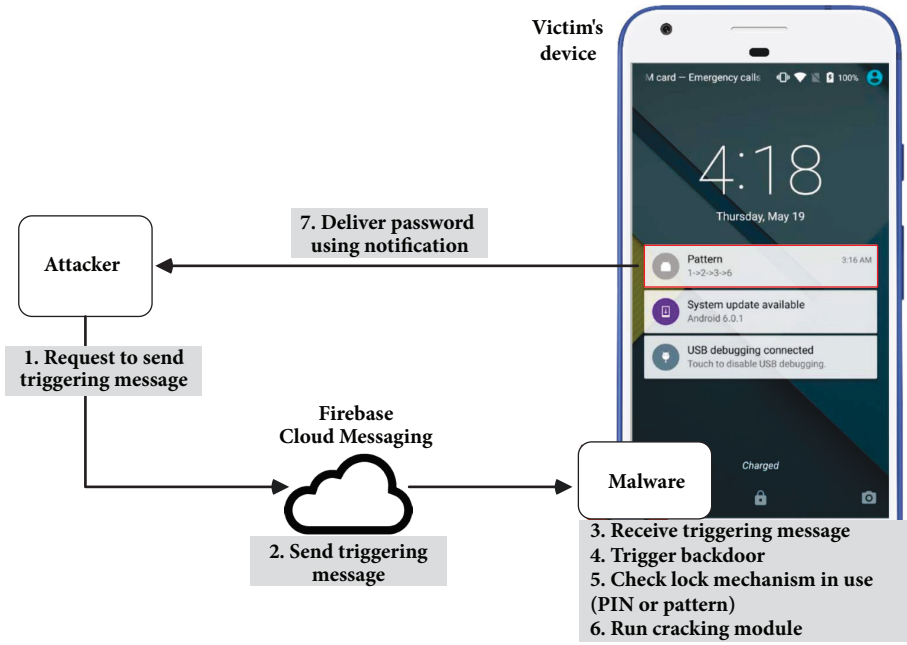


Figure 3: Attack procedure of push notification-based backdoor.

```
(1)  passwdHash ⟵ readPasswdHash()
(2)  oldPasswdHash ⟵ readOldPasswdHash()
(3)  if passwdHash = oldPasswdHash then
(4)      oldPasswd ⟵ readOldPasswd()
(5)      showNotification(oldPasswd)
(6)  else
(7)      randomSalt ⟵ readSalt()
(8)      while passwdToTest ⟵ readDictionary() do
(9)          saltedPasswd ⟵ passwdToTest ‖ randomSalt
(10)         if computeHash(saltedPasswd) = passwdHash then
(11)             showNotification(passwdToTest)
(12)             break
```

Algorithm 1: Password guessing algorithm.

To improve the efficiency of the password cracking procedure, the attacker can build a dictionary from real datasets of PINs (or patterns) and preconfigure the backdoor application with the dictionary. Algorithm 1 briefly summarizes our dictionary attack procedure. To avoid unnecessary computations, first of all, the password cracker checks if the victim's password has been updated by comparing the hash value of the old password found previously (`oldPasswdHash` in line (2)) with the current hash value (`passwdHash` in line (1)) read from `password.key` or `gesture.key` file. Only when `passwdHash` is different from `oldPasswdHash`, which means that the victim's password has been updated, does the password cracker proceed to the dictionary attack procedure (in lines (7)–(12)).

In the dictionary attack procedure, the password cracker first reads the salt from `locksettings.db` file only if the PIN is used by the victim (in line (7)). On the other hand, no salt is used for the pattern lock mechanism; thus the password cracker skips line (7) for patterns. The password cracker then reads the candidate passwords to test one by one from the PIN or pattern dictionary (in line (8)). If the candidate password is a PIN, then it is concatenated with the salt (in line (9)). For each candidate password, the password cracker computes the hash value of the candidate by calling *computeHash()* (in line (10)) and then compares the computed hash value with the hash value read from `password.key` or `gesture.key` file (in line (10)). If both hash values are the same, this means that the candidate password that has been tested is the victim's password. The internal process of *computeHash(input)* differs depending on whether the given input is a PIN or a pattern. If the input is a PIN, *computeHash(input)* returns *SHA-1(input)‖ MD5(input)*. Otherwise, if the input is a pattern, *computeHash(input)* returns *SHA-1(input)*.

Unlike 4-digit PINs, the Android pattern lock mechanism does not require a random salt, and this allows attackers to precompute and use the dictionary of all possible patterns to

shorten the time to guess a victim's pattern. However, for all 389,112 possible patterns, the dictionary file size is approximately 5.2 Mbytes, and embedding such a large dictionary file in the APK file of a legitimate application causes a significant increase in the size of the APK file. This significant change on the APK file could be a clue for detecting the backdoor. For such reasons, it is not recommended to use the dictionary of screen lock patterns.

After finding out the victim's password, the backdoor application finally informs the attacker of the password by showing up a notification popup (Step 7). A video demo of our push notification-based backdoor is available at (https://youtu.be/iyWhsYyPFnc).

*5.3.2. DDoS Bot.* DDoS bot application is designed to perform DDoS attacks against target servers and connected with the triggering application (controlled by the botmaster) through push notification services. Whenever the botmaster wants to launch a DDoS attack on a target server, the botmaster sends push messages to his (or her) bot applications via FCM servers. These push messages may contain some information necessary for DDoS attacks such as the network identifier of the target server and the transmission rate of attack traffic as well as a command to trigger the bot applications. Once receiving this push message from the triggering application, each DDoS bot application is automatically triggered and extracts the information of attack from the received push message. Based on the information retrieved from the push message, the bot application performs DDoS attacks against the target server.

## 6. Evaluation

This section shows the feasibility of push notification-based malware through our backdoor implementation. Specifically, we analyzed the stealthiness of the backdoor using commercial antimalware tools and also measured the execution time and power consumption of the backdoor.

*6.1. Stealthiness of Push Notification-Based Malware.* When a casual user encounters the installed backdoor on his (or her) Android device, it is difficult for him (or her) to notice that the backdoor is being used, since our backdoor never changes his (or her) unlock password. In addition, our backdoor is *temporarily* executed only when an attacker sends a specific push message to the backdoor.

To validate the resilience of our backdoor against malware detection, it is important to see whether our backdoor implementation could be detected by existing antimalware tools. For this validation, we analyzed the APK file of our backdoor application with several popular antimalware tools (e.g., VirusTotal [5] and SandDroid [6]). For a comparison purpose, we also analyzed the APK files of other normal applications under the same conditions.

We first analyzed the APK file of our backdoor application using 55 different types of antimalware scanners provided by VirusTotal [5]. However, all the scanners failed to classify our backdoor application as malware. This web page (https://goo.gl/5vOeuG) shows the detailed results of our analysis using VirusTotal.

SandDroid [6] provides both static and dynamic analyses of given Android APK files and returns the estimated risk score of the APK files in the range of 0 to 100; 0 and 100, respectively, represent the lowest and the highest level of risk. In this second experiment, we analyzed the APK file of our backdoor application using SandDroid, and the resulting risk score was 24. To understand the risk level of this score (24), we randomly selected 30 Google applications available on the Google Play Store and analyzed the APK files of all those applications in the same manner. The average risk score of all those applications was 26.47, which is very close to that of our backdoor application, and the standard deviation was 17.87. We also performed the same analysis for 30 malware samples. Unlike our backdoor application, their risk scores were all 100. From these results, we concluded that the existing antimalware scanners were ineffective to detect our backdoor application.

Repackaging malware in a legitimate application is a popular method to distribute Android malware [16]. If repackaging our backdoor causes significant changes on the original APK file, such changes could be a clue for detecting the backdoor. To see the impact of such repackaging on an APK file, we repackaged our backdoor in the APK file of a legitimate application that originally uses Firebase and investigated changes in the APK file in several aspects: required permissions, file size, total number of contained files, services, activities, intent filters, and so forth. In summary, the dictionary files and codes of the backdoor caused an increase in the APK file size by about 1.5 Mbytes and an increase in the number of contained files by 8. But, except these changes, no other changes were observed in the APK file.

*6.2. Execution Time.* The execution time of our backdoor is important because the longer the execution time is, the more likely it is to be detected. Thus we measured the execution time of the backdoor application by conducting both brute-force and dictionary attacks for PIN and screen lock pattern. We used existing real-world datasets of 4-digit PINs [24] and patterns [25]. The PIN dataset consists of 204,508 PINs collected from that number of iPhone users in 2011, and the pattern dataset consists of 389,112 patterns generated from the patterns collected from 312 Android users in 2015 using the 3-gram Markov model [25]. We used each of the entire datasets as a PIN and pattern dictionary for our dictionary attack. For measuring the execution time of our backdoor application, we randomly selected 5,000 PIN (or pattern) samples from the entire dataset of PINs (or patterns) and measured the average execution time (i.e., the average time taken to identify the lock mechanism and crack the tested PIN (or pattern)) of the backdoor application after it has been triggered by a push message.

Table 1 summarizes the experiment results. For 4-digit PINs, the dictionary attacks cracked 52.68% of all the tested PINs within 1 second, whereas the brute-force attacks cracked only 10.60%. The dictionary attacks show on average 1.5 times faster execution time than the brute-force attacks. For screen lock patterns, the dictionary attacks cracked 85.34% of all the

Table 1: Average password cracking time (B: brute-force attack, D: dictionary attack).

| Time (sec) | 4-digit PIN | | Lock pattern | |
|---|---|---|---|---|
| | B | D | B | D |
| **<1** | 10.60% | 52.68% | 44.10% | 85.34% |
| **<2** | 57.88% | 74.88% | 62.36% | 93.76% |
| **<3** | 70.98% | 85.76% | 68.90% | 96.22% |
| **<4** | 83.72% | 93.34% | 71.00% | 97.54% |
| **<5** | 94.28% | 98.72% | 76.60% | 98.28% |
| **Avg.** | 2.3 s | 1.5 s | 5.0 s | 1.0 s |

Table 2: Power consumption (J) of backdoor application and ratio (%) of how much of the total power usage is consumed by backdoor application (STD: standard deviation).

| | Changed | | Not changed | |
|---|---|---|---|---|
| | **Pattern** | **PIN** | **Pattern** | **PIN** |
| **Avg.** | 74.597 J | 10.393 J | 0.043 J | 0.050 J |
| **STD** | 4.493 J | 1.944 J | 0.056 J | 0.050 J |
| Ratio | 10.862% | 1.669% | 0.007% | 0.008% |

tested unlock patterns within 1 second, whereas the brute-force attacks cracked only 44.10%. The brute-force attacks show on average 4.93 times slower execution time than the dictionary attacks.

*6.3. Power Consumption.* There exist several attempts to detect mobile malware by monitoring the power consumption patterns of applications [26–28]. For example, if our backdoor application consumes an excessive amount of battery power deviated from typical power usage, this would make our backdoor detected easily by antimalware systems based on power consumption monitoring. Thus, we performed experiments to investigate the power usage of our backdoor application. In these experiments, PowerTutor [29] was used to measure the power consumption of the backdoor. To see the impact of the password cracking process only, we compared the power consumption of the backdoor in two different cases; `Changed` and `Not changed`. In `Changed` case, the password is changed for every execution, and thus the password cracking process in the backdoor application is always executed. On the other hand, in `Not changed` case, the password is never changed, and so the only thing the backdoor application does is to show up a notification of the previously cracked password without performing the password cracking process.

Table 2 shows the average power consumption of the backdoor application and the ratio of how much of the total power usage is consumed by the backdoor. For 4-digit PINs, the backdoor application consumed only a small fraction of the total power usage in both cases of `Changed` and `Not changed`, whereas, for screen lock patterns, the backdoor application had a considerable impact on power consumption, especially in `Changed` case. In general, however, it is uncommon for users to frequently change the unlock passwords of their devices.

## 7. Push Notification-Based Malware Prevention

This section discusses possible defense strategies to mitigate push notification-based malware.

*7.1. Filtering Registration Requests.* The first line of defense against push notification-based malware is to prevent malicious applications from being registered into FCM servers. In order to achieve this, when receiving a registration request from a subscriber application, a FCM server should be able to determine whether the requesting application is suspicious or not and then stop the registration of the subscriber application if it is regarded as suspicious. This defense might be effective because push notification-based malware whose registration request has been denied by a FCM server is unable to receive any message or command from the attacker.

To detect suspicious applications, a FCM server can take advantage of a reputation system of mobile applications [30]. As an example, the reputation system counts how often a given application is observed in a large group of FCM users, and uncommon applications may be tagged as suspicious. If a given application, requesting a subscription for push notifications, is considered as suspicious according to the reputation system, then the FCM server denies the registration request. In this way, we can effectively block suspicious applications from receiving messages through push notification services.

In Figure 4, `Registration Manager` in the FCM server inspects registration requests incoming from subscriber applications on user devices and issues registration tokens only to applications that pass the checks through the reputation system.

*7.2. Filtering Token Requests.* In order for attackers to send push messages to their malicious applications, the attackers

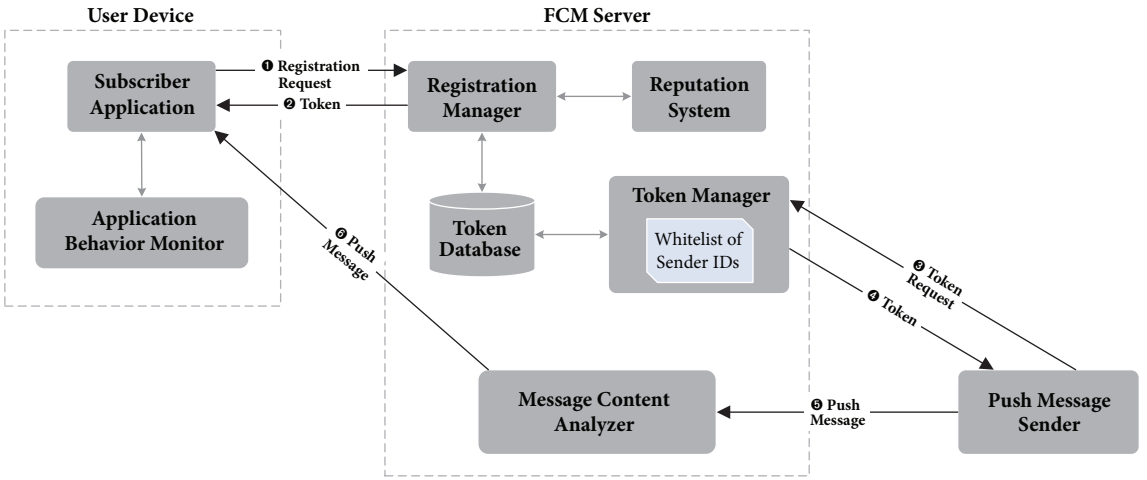


Figure 4: System architecture to mitigate push notification-based malware.

should first obtain the registration tokens of those applications. That is, if we prevent the attackers from obtaining the registration tokens of their malicious applications, we can disconnect the communication channels based on push notification between the attackers and their malicious applications.

In most push notification services including FCM, subscriber applications themselves forward their registration tokens to application servers after obtaining the tokens from the FCM server. That is, malware can easily forward the token to the attacker if the malware obtains the registration token.

A possible option to prevent this is not to allow subscriber applications to send out their tokens. Instead, the FCM server should manage the registration tokens of subscriber applications in a centralized manner and control the accesses of the tokens from application servers. In order to block the tokens stored in the user devices from being sent out, we need a way to detect and block attempts to send tokens outside, and techniques such as TaintDroid [31] can be utilized for this objective.

Senders who want to send push messages to their applications should first request the FCM server for the tokens of the applications, and the request contains the identifier of the sender. Once receiving the request for a token, the FCM server determines whether or not it can provide the token to the sender. For this filtering decision, the FCM server can maintain a whitelist of *trusted* sender IDs. That is, if a received token request is from a sender ID that is not included in the whitelist, the FCM server denies the request.

Although blacklisting is effective in preventing known malicious applications, it can be weak against preventing new ones. In practice, it is difficult to efficiently update the central database of malicious subscriber applications. Whitelisting, on the other hand, is effective against unknown malicious subscriber applications since only those considered 'trusted' are allowed to send push messages. Therefore, our recommendation would be to use whitelisting rather than blacklisting.

In Figure 4, `Token Manager` receives token requests from push message senders, searches the whitelist for the senders' IDs, and delivers the requested token only to the whitelisted senders.

*7.3. Analyzing Push Message Contents.* Another defense strategy is to detect malicious push messages by analyzing the contents of push messages. To avoid permission-based detection mechanisms, an attacker may repackage his or her malware in a legitimate application that originally uses a push notification service. Let $app_x$ and $app'_x$ denote the normal version and the malicious repackaged version of application $x$, respectively. From the attacker's perspective, it is important to minimize the difference between $app_x$ and $app'_x$ to avoid detection. Therefore, in many cases, the attacker is unlikely to modify the original behaviors of $app_x$ while embedding the malicious logic into $app_x$. Then $app'_x$ receives push messages from both the legitimate senders and the attacker. Under this condition, the malicious logic embedded in $app'_x$ should be triggered only for push messages from the attacker while keeping silent for push messages from the legitimate senders.

Let us assume that the backdoor is repackaged from $app_x$. If the backdoor launches its password cracking activity even for push messages from the legitimate senders, the notification popup of the password shown by the backdoor could be found by the victim and this situation would increase the chance for the victim to detect the backdoor. To prevent this, the backdoor should be able to distinguish the attacker's push messages from other push messages from the legitimate senders so that the password cracking activity is launched only when it has been requested by the attacker. To achieve this, a push message from the attacker should include a specific pattern so that the malware can identify the attacker's command. In this case, it is possible to detect the attacker's messages by checking the existence of the specific patterns on received push messages.

In Figure 4, `Message Content Analyzer` in the FCM server inspects the contents of messages requested for push notifications by senders and filters out messages that exhibit malicious patterns.

We implemented a proof-of-concept of `Message Content Analyzer` which performs analysis of the contents of

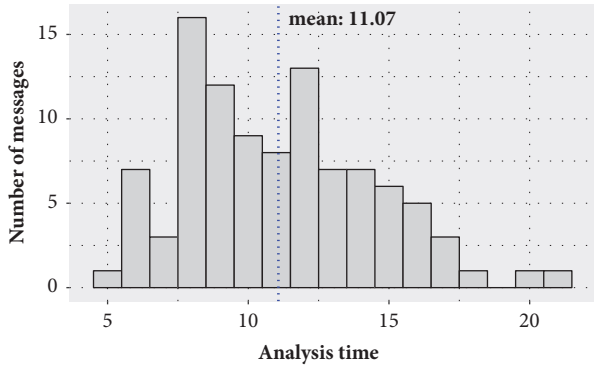


Figure 5: Distribution of the analysis times observed for 100 push messages (the blue dotted line shows the average analysis time).

push messages with a rule-based model. We assume that the analyzer has a database of malicious patterns. When the analyzer receives a push message, it verifies that each malicious pattern defined in the database is included in the received push message. If the received push message contains any of the malicious patterns defined in the database, the analyzer drops the message so that the message is not delivered to the user device.

To show the feasibility of `Message Content Analyzer`, we evaluated our proof-of-concept with 100 push messages by measuring the time taken to analyze those messages. For this evaluation, we used Google Pixel 2 with Qualcomm Snapdragon 835, 4 GB RAM, and Android 8.0 Oreo. On average, our analyzer took 11.07 milliseconds for analyzing each message (standard deviation was 3.37 milliseconds). Figure 5 shows the distribution of the analysis times observed for the 100 messages. The $x$-axis shows the observed analysis times ranging from 5 to 21 milliseconds, and the $y$-axis shows the number of messages showing each analysis time in that range. The most frequently observed analysis time was 8 milliseconds, and the analysis times of 72 out of 100 messages fell into the range of 8 to 14 milliseconds.

*7.4. Machine Learning-Based Defense Strategy.* To improve the effectiveness of suspicious push message detection, machine learning techniques can be utilized. To achieve this goal, we can build a classifier with a set of features that can be collected by the FCM server for sending a push message. Naturally, it is critical to choose proper features of push messages that can be used to distinguish the push messages for malware from push messages for normal applications.

A promising candidate feature is the number of recipient applications of a push message because the number of applications receiving the attack messages is typically much smaller than the number of applications receiving the normal messages. Also, push message size, interarrival time, and keywords used in push messages can be used as reasonable features for our purpose.

*7.5. Monitoring Application Behaviors.* Another defense strategy against push notification-based malware is to monitor and analyze an application's behaviors after the application receives push messages. If an application exhibits behaviors similar to those of known malware or if an application's behaviors deviate from normal behavior patterns, then we can consider the application as abnormal application and take appropriate counteractions.

Our backdoor prototype, for example, has the following behavior patterns: after being triggered by a push message received from the triggering application, the backdoor application first accesses the password hash file, performs computationally intensive tasks, and then pops up a notification. If some application exhibits such behaviors, then we can suspect the application as backdoor. As another example, we can consider a push message-based bot application [1, 2]. This bot application typically transmits a large amount of traffic to a specific host after being triggered by the received push message. In other words, it shows a behavior characteristic that transmits a very large number of messages after receiving a few push messages. Therefore, we can detect the push message-based bot application by monitoring the application's transmission patterns after receiving a push message.

*7.6. Reviewing and Analyzing Security Sensitive Events.* Another mitigation strategy is to keep the history of all security-sensitive events that have occurred on a user device for later investigation. For this, Google or a device manufacturer's server can collect security-sensitive events from user devices, and the device owners who want to know what happened on their devices can review the collected events by visiting the server website. Reviewing the history of suspicious events increases the chance of detecting the malware.

Another possible approach is to collect and analyze diverse types of events occurring on a user device using machine learning algorithms. Through this analysis, usual patterns of events observed on the user device can be identified. By monitoring the deviation from the usual patterns, it is possible to automatically detect suspicious events.

## 8. Conclusions

We presented a design of push notification-based malware which allows attackers to remotely trigger and control malware on victims' devices. We also implemented a backdoor prototype based on our malware design on Android devices and evaluated its stealthiness and feasibility. The evaluation results showed that our malware implementation could effectively evade existing malware detection tools. In addition, our backdoor successfully cracked about 98% of all the tested PINs and patterns in 5 seconds with only a fraction (less than 0.01%) of the total power consumption of the device. Finally, we proposed several defense strategies to mitigate push notification-based malware by carefully analyzing its attack process.

## Data Availability

The source codes of our implementation and the data files of the evaluation results are available at a GitHub repository (https://github.com/rymuff/os-data-availability). The GitHub repository contains the URL links to the source codes

of the triggering application and the backdoor application. The repository also contains the URL links to the following data files of the evaluation results: the data files of the analysis results of the backdoor application using VirusTotal and SandDroid and the data files of the execution time and power consumption measurements of the backdoor application.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This research was supported by the MSIT (Ministry of Science and ICT), Republic of Korea, under the ITRC (Information Technology Research Center) support program (IITP-2018-2015-0-00403) supervised by the IITP (Institute for Information & communications Technology Promotion) and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2016R1A6A3A11930593).

## References

[1] S. Zhao, P. P. Lee, J. C. Lui, X. Guan, X. Ma, and J. Tao, "Cloud-based push-styled mobile botnets: a case study of exploiting the cloud to device messaging service," in *Proceedings of the the 28th Annual Computer Security Applications Conference*, pp. 119–128, Orlando, FL, USA, December 2012.

[2] H. Lee, T. Kang, S. Lee, J. Kim, and Y. Kim, "Punobot: mobile botnet using push notification service in android," in *Proceedings of the International Workshop on Information Security Applications*, pp. 124–137, Springer, 2013.

[3] W. Chen, X. Luo, C. Yin, B. Xiao, M. H. Au, and Y. Tang, "Muse: Towards robust and stealthy mobile botnets via multiple message push services," in *Australasian Conference on Information Security and Privacy*, pp. 20–39, Springer, 2016.

[4] J. Cho, G. Cho, S. Hyun, and H. Kim, "Open sesame! design and implementation of backdoor to secretly unlock android devices," *Journal of Internet Services and Information Security*, vol. 7, no. 4, pp. 35–44, 2017.

[5] V. Total, "VirusTotal-Free online virus, malware and URL scanner," https://www.virustotal.com.

[6] B. R. Team et al., "SandDroid: An APK Analysis Sandbox. Xian Jiaotong University," https://sanddroid.xjtu.edu.cn.

[7] Y. Zeng, K. G. Shin, and X. Hu, "Design of SMS commanded-and-controlled and P2P-structured mobile botnets," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '12)*, pp. 137–148, April 2012.

[8] J. Hua and K. Sakurai, "A SMS-based mobile botnet using flooding algorithm," in *IFIP International Workshop on Information Security Theory and Practices*, pp. 264–279, Springer, 2011.

[9] C. Mulliner and J.-P. Seifert, "Rise of the iBots: owning a telco network," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software (Malware '10)*, pp. 71–80, IEEE, October 2010.

[10] M. Anagnostopoulos, G. Kambourakis, and S. Gritzalis, "New facets of mobile botnet: architecture and evaluation," *International Journal of Information Security*, vol. 15, no. 5, pp. 455–473, 2015.

[11] G. Andre and P. Ramos, "Boxer sms trojan," Technical Report, ESET Latin American Lab, Bratislava, Slovakia, 2013.

[12] E. Messmer, "Backdoor.androidos.obad.a," *Network World*, Oct. 2013, (Accessed on Jan. 30, 2018), http://contagiominidump.blogspot.in/2013/06/backdoorandroidosobada.html=0pt.

[13] K. Singh, S. Sangal, N. Jain, P. Traynor, and W. Lee, "Evaluating Bluetooth as a medium for botnet command and control," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 61–80, Springer, 2010.

[14] T. Yin, Y. Zhang, and S. Li, "DR-SNBot: a social network-based botnet with strong destroy-resistance," in *Proceedings of the 9th IEEE International Conference on Networking, Architecture, and Storage (NAS '14)*, pp. 191–199, IEEE, August 2014.

[15] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning, "Andbot: towards advanced mobile botnets," in *Proceedings of the 4th USENIX Conference on Large-Scale Exploits and Emergent Threats*, p. 11, USENIX Association, 2011.

[16] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY '12)*, pp. 317–326, ACM Press, San Antonio, TX, USA, Feburary 2012.

[17] S. Grzonkowski, A. Mosquera, L. Aouad, and D. Morss, "Smartphone security: an overview of emerging threats," *IEEE Consumer Electronics Magazine*, vol. 3, no. 4, pp. 40–44, 2014.

[18] I. Muslukhov, Y. Boshmaf, C. Kuo, J. Lester, and K. Beznosov, "Know your enemy: the risk of unauthorized access in smartphones by insiders," in *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services (MobileHCI '13)*, pp. 271–280, ACM Press, Munich, Germany, August 2013.

[19] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*, pp. 95–109, San Francisco, Calif, USA, May 2012.

[20] A. Ludwig and M. Mille, "Diverse protections for a diverse ecosystem: Android security 2016 year in review," https://goo.gl/6o4tBf, March 2017, https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf=0pt.

[21] F. Howarth, "Is rooting your phone safe? the security risks of rooting devices," https://goo.gl/axbkX9, October 2015, https://insights.samsung.com/2015/10/12/is-rooting-your-phone-safe-the-security-risks-of-rooting-devices=0pt.

[22] A. Boxall, "80% of android phone owners in china have rooted their device," https://goo.gl/dH4zQZ, April 2015, http://www.businessofapps.com/80-android-phone-owners-china-rooted-device=0pt.

[23] H. Zhang, D. She, and Z. Qian, "Android root and its providers: a double-edged sword," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pp. 1093–1104, Denver, CO, USA, October 2015.

[24] H. Kim and J. H. Huh, "PIN selection policies: are they really effective?" *Computers & Security*, vol. 31, no. 4, pp. 484–496, June 2012.

[25] Y. Song, G. Cho, S. Oh, H. Kim, and J. H. Huh, "On the Effectiveness of Pattern Lock Strength Meters," in *Proceedings of the the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*, pp. 2343–2352, Seoul, Republic of Korea, April 2015.

[26] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proceedings of the*

*6th International Conference on Mobile Systems, Applications, and Services ( MobiSys '08)*, pp. 239–252, Breckenridge, CO, USA, June 2008.

[27] T. K. Buennemeyer, T. M. Nelson, L. M. Clagett, J. P. Dunning, R. C. Marchany, and J. G. Tront, "Mobile device profiling and intrusion detection using smart batteries," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS '08)*, p. 296, Waikoloa, HI, USA, January 2008.

[28] L. Liu, G. Yan, X. Zhang, and S. Chen, "VirusMeter: preventing your cellphone from spies," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 244–264, Springer, 2009.

[29] L. Zhang, B. Tiwana, Z. Qian et al., "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the 16th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '16)*, pp. 105–114, Scottsdale, AZ, USA, October 2010.

[30] Z. Ramzan, V. Seshadri, and C. Nachenberg, "Reputation-based security: an analysis of real world effectiveness," *Symantec Corporation*, 2009.

[31] W. Enck, P. Gilbert, S. Han et al., "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, p. 5, 2014.