Real-Time Detection and Prevention of Android SMS Permission Abuses

Weiliang Luo
Dept. of Computer Science
UT San Antonio
wluo@cs.utsa.edu

Shouhuai Xu
Dept. of Computer Science
UT San Antonio
shxu@cs.utsa.edu

Xuxian Jiang
Dept. of Computer Science
NC State University
jiang@cs.ncsu.edu

ABSTRACT

The Android permission system informs users about the privileges demanded by applications (apps), and in principle allows users to assess potential risks of apps. Unfortunately, recent studies showed that the installation-time permission verification procedure is often ignored, due to users' lack of attention or insufficient understanding of the privileges or the Android permission system. As a consequence, *malicious* apps are likely granted with security- and privacy-critical permissions, and launch various kinds of attacks without being noticed by the users. In this paper, we present the design, analysis, and implementation of DroidPAD, a novel solution that aims to leverage system-wide state information to detect and block in real-time possible abuses of Android permissions. Especially, with a focus on SMS-related permissions, we have implemented a proof-of-concept prototype. Our evaluation based on 48 representative benign and malicious apps shows that DroidPAD successfully detected SMS permissions-abusing activities with low false-positive rates, and zero false-negative rates.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Invasive software*

Keywords

Mobile Application; Smartphone Security; Permission Abuse;

1. INTRODUCTION

Android is leading the smartphone market, perhaps because of the hundreds of thousands of attractive third party applications (apps). Unfortunately, the success of Android platform has also attracted various attackers, especially malware writers who write malicious apps to gain financial profit or collect private data. Such malicious apps often exploit the weakest link in the system, namely the human users who often install and run apps without carefully examining the potential risk. This is true despite that Android provides a *permission system*, which aims to help users assess the potential risks of apps. The permission system forces each app to declare the permissions (privileges) it wants, and gives the users the opportunity to decide whether to install the app or not. While sound in principle, the permission system has been largely ignored by human users because the permissions and policies are complex and hard to understand. According to a recent survey [6], only 17% participants pay attention to permissions when installing apps and 42% laboratory participants (total of 25) are not even aware of the existence of permissions. As a consequence, malicious apps often can get the permissions they wanted, which might have contributed to the explosive growth of Android malware [13].

There have been two approaches to addressing the above *permission-abuse* problem. The first approach is to give the users more management/control power, such as: allowing the users to assign and revoke permissions of apps after the installation [8], enforcing usage control with user-defined policies [1], allowing the users to better control the access to their private data [15]. However, this approach does not strive to help the users deal with permission-abuse before they decide to revoke some permissions. Moreover, this approach still largely requires the users to have a good understanding of the Android permission system, which, if true, may have already prevented the users from installing potentially malicious apps in the first place. The second approach is to identify malicious apps in mobile app marketplaces (e.g., [3, 14, 7, 12]) and then alert the users not to install them. However, this approach does not deal with the malicious apps that are already installed.

In this paper, we investigate a new approach to addressing the permission-abuse problem. We aim to detect and block permission-abuses in real time, so as to mitigate the damage of malicious apps that are already installed. This approach does not force the users to understand the complex Android permission system and strives to be as transparent as possible to the users. We present the design, implementation and analysis of a such system, called AnDroid Permission Abuses Detector (DroidPAD). The DroidPAD presented in this paper deals with the abuses of SMS permissions, by intercepting operations of interest, collecting three types of state information relevant to the operations (i.e., phone state, app state, and user input), analyzing the usage scenarios of permissions, and identifying and blocking permission abuses. In order to facilitate the first two steps that require the system privilege, we slightly modified the Android framework. Experiment results show that it can detect and block abuses of SMS permissions with low false-positive rates (2.1% for `SEND_SMS`, 8.3% for `RECEIVE_SMS`, 0% for `WRITE_SMS`) and zero false-negative rates. Performance evaluation demonstrates that the overhead is almost negligible.

The rest of this paper is organized as follows. Section 2 describes the design of DroidPAD. Section 3 presents the implementation of DroidPAD. Section 4 discusses the effectiveness and performance evaluation results. Section 5 reviews related prior work. Section 6 examines the limitation of DroidPAD and discusses the future research directions. Section 7 concludes the paper.

2. SYSTEM DESIGN

Design and implementation of the envisioned full-fledged DroidPAD is a challenge because of the large number of apps. As a proof of concept, we in this paper present our first step towards the ultimate goal, by focusing on addressing the abuse of SMS permissions in real time.

2.1 Threat Model and Assumptions

In the threat model, Android users are assumed to be honest, but apps are potentially malicious and may abuse their granted permissions to conduct attacks. We assume that all malicious apps are written by the most powerful attacker. Since our security enhancement needs to reside at the Android framework layer, we assume that the extended Android framework and the underlying Linux kernel are secure throughout their entire life-cycle. This assumption implies that the attacker cannot gain root privileges of the target Android devices, and that the apps use their own permissions that are granted by the human users because malicious apps cannot compromise the other benign apps (implied by the security of the Android framework).

2.2 Design Objectives

The ultimate goal of DroidPAD is to detect and block all abuses of permissions in real time. It is meant to be an indispensable component in any comprehensive solution to securing Android smartphones. Its design objectives include three aspects: security, usability, and performance. First, the security objective is to use some pre-defined models to detect and block permission-abuse in real time. This requires to ensure the followings: *data origin integrity*, meaning that sources of the data collected by DroidPAD cannot be forged by the attacker; *data integrity*, meaning that the collected data cannot be manipulated by the attacker; *data secrecy*, meaning that the collected data is kept confidential from the attacker; *code integrity*, meaning that its own code base cannot be manipulated by the attackers. Second, the usability objective is that it neither forces the users to understand the complex permission system, nor requires much significant (if any) involvement of users in managing/using it. In other words, it should be as transparent as possible to the users. Third, the performance objective it that it should impose as little as possible performance overhead.

2.3 System Overview

The key insight behind DroidPAD's design for detecting and blocking abuses of SMS permissions is to leverage certain system state information to determine whether a permission usage is benign or malicious. At a high level, DroidPAD is achieved via two core components. The first core component is called *offline usage scenario profiling*. For a targeted set of permissions, we manually analyze the usage scenarios of both known malicious and benign apps that use these permissions. In so doing, we can define malicious and benign permission usage scenarios by selecting the appropriate system state information. The second core component is called *online permission abuse detection*, which utilizes the pre-defined malicious and benign permission usage scenarios to identify operations that abuse permissions. In order to intercept the operations and collect the relevant system state information, we need to modify the Android framework slightly. These components are elaborated below.

2.4 Offline Usage Scenario Profiling

In order to differentiate the malicious permission usage scenarios from the benign ones, we studied behaviors of some popular/benign and known malicious SMS apps. The benign apps are: `Go SMS`, `Auto SMS`, and `stock SMS app` in Android 4.12. The malicious apps belong to the following malware families: `FakePlayer`, `Walkinwat`, `HippoSMS`, `JiFake`, `Zsone`, and `OpFake`. In the profiling process, we manually ran these apps on a modified Android image to log their behaviors of sending SMS messages and writing the SMS database. Table 1 summarizes the observed usage scenarios corresponding to three SMS permissions — `SEND_SMS`, `RECEIVE_SMS` and `WRITE_SMS` — that are widely abused by Android malwares [13]. For example, `SEND_SMS` is attractive to Android malwares because it can be abused to send premium number SMS messages for financial gains; `RECEIVE_SMS` can be abused to block incoming SMS messages silently without being noticed by the user; `WRITE_SMS` can be abused to write arbitrary contents into the SMS database and therefore launch the "smishing" attack [10].

For the `SEND_SMS` permission, we identify five benign usage scenarios: Normal SMS (*BS1*), Scheduled SMS (*BS2*), Auto reply SMS (*BS3*), Auto forward SMS (*BS4*), and Signature-appended SMS (*BS5*). The first four scenarios are self-explaining, and the last scenario is that a SMS message is sent with a string appended at the end (e.g., a message text body is "OK – from Alice" while the user actually typed "OK" only). All these benign usage scenarios involve user input at some point. On the other hand, the malicious permission usage scenario *MS1* is unrelated to the user input behavior. This allows us to detect abuses of the `SEND_SMS` permission by examining whether or not there was an user input, which requires to to check some system state information as follows. For scenario *BS1*, we need to check *current user input*, the system state information that is generated by the current subject application. In order to enhance detection accuracy, we further propose to check the following system state information: *screen state* (i.e., screen on/off), *lock state* (i.e., screen locked/unlocked), *application uptime* (i.e., elapsed time since application started) and *application visibility* (i.e., foreground/background application). They are useful because, for example, the average *application uptime* in our experiment for sending the first SMS message is 0.947 seconds for malicious apps (*MS1*) and 5.26 seconds for benign apps. For scenarios *BS2-BS5*, we need to capture past user input activities that were maintained for configuration purposes (i.e., *purpose specific user input*). We also propose to use the *telephony state* to help identify *BS3* or *BS4* because both scenarios can be triggered only by telephony state change (e.g, newly received SMS or phone call).

For the `RECEIVE_SMS` permission, we identify two benign

Table 1: Usage scenarios "XYa" of three SMS-related permissions SEND_SMS (S for short), RECEIVE_SMS (R for short) and WRITE_SMS (W for short), where $X \in \{B, M\}$ indicates the scenario is benign (B) or malicious (M), $Y \in \{S, R, W\}$ indicates the type of permission, and a indicates the index of the scenario.

SEND_SMS	*BS1*: User composes the SMS body and sends it instantly using the UI of the subject app.
	BS2: User composes the SMS body and schedules the time when SMS should be sent. Then, the subject app will send SMS out automatically at that time point.
	BS3: User composes the SMS body and configures the subject app to automatically send that SMS as a response to an incoming SMS or phone call when he/she is busy (e.g., meeting, driving).
	BS4: User inputs the forward number and configures the subject app to automatically forward every incoming SMS to that number.
	BS5: User configures the subject app to send SMS with appended signature.
	MS1: One or more SMS is sent out to unknown number with unknown content, at the time point determined by malware writer.
RECEIVE_SMS	*BR1:* The subject app intercepts the newly received SMS and write into SMS database.
	BR2: User configures rules for unwanted SMS, based on which the subject app filters the incoming SMS silently.
	MR1: The subject app filters the incoming SMS unknowingly to user.
WRITE_SMS	*BW1:* The subject app writes received SMS message as incoming SMS into database.
	BW2: The subject app writes a new chat message received from internet as incoming SMS into database.
	BW3: The subject app writes backuped SMS messages into database as requested by user.
	MW1: The subject app writes deceptive content as incoming SMS message into the database.

usage scenarios, *BR1* and *BR2*, and one malicious usage scenario *MR1*. This is based on the observation that every new SMS message should be written into the SMS database (*BR1*) unless SMS is blocked (*BR2, MR1*). This is reasonable because benign apps will block SMS only when the user defined a policy and the new SMS message is included; whereas, malicious apps block messages as commanded by attackers. This suggests us to accommodate user-defined blocking policy, which is another kind of purpose-specific user input.

For the WRITE_SMS permission, we need to know when WRITE_SMS is used to insert an *incoming* SMS message, because the damage of inserting other types (e.g., *draft, sent*) of messages is still unclear. Accordingly, we identify three benign scenarios: *BW1, BW2* and *BW3*. Although we do not find any existing app that abuses this permission, there is a relevant emerging attack called "smishing" (by abusing the WRITE_SMS permission), which motivates us to define the malicious usage scenario *MW1*. In order to differentiate the malicious scenarios from the benign ones, we exploit the following system state information: *SMS state*, which indicates whether or not there is a new SMS message received at a given time span (1 minute in our case). This information uniquely identifies the benign scenario *BW1*, which is the most common usage scenario of WRITE_SMS. In order to differentiate *BW2, BW3* from *MW1*, which a non-trivial task because both the receiving of chat messages (from Internet) and the restoring SMS message are application-specific and cannot be captured at the Android system level, we propose to exploit the UID-based isolation on the centralized SMS database. This allows us to differentiate *MW1* from *BW2*. In order to uniquely identify *BW3*, we propose to ask for the user's confirmation; this is the only place the human user is involved, slightly though.

2.5 Online Permission-Abuse Detection

As shown in Figure 1, the online permission abuse detection system consists of five components: *Permission Usage Monitor*, *State Information Monitor*, *Permission Abuse Detector*, DroidPAD *Manager*, and *Configuration Database*. These components are highlighted below and their concrete implementations are described in Section 3.

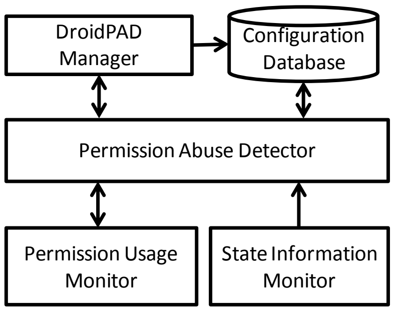

Figure 1: Online permission-abuse detector

The permission usage monitor is a privileged component that can intercept, suspend, and terminate the use of permissions. State information monitor is a combination of multiple sensors that are bound to various types of state information mentioned above, such as *screen state*, *application visibility*, and *user input.* Various kinds of state information are sent to the permission abuse detector, whenever a permission usage is intercepted. To differentiate a permission abuse from a benign use, the permission abuse detector relies on the collected state information to identify the permission usage as one of the pre-defined scenarios. The configuration database is the central storage for the system, containing information such as *purpose-specific user input* and *configured exception.* Once a permission abuse is detected, the permission usage monitor will terminate the target permission usage and an alert message will be prompted by the DroidPAD manager to the human user.

3. IMPLEMENTATION

We implemented a prototype DroidPAD system based on Android source code version 4.12. In what follows we describe the implementation of the five components of the online permission-abuse detection system (Figure 1).

3.1 Permission Usage Monitor

The core function of this component is to intercept permission usages. For intercepting usage of the `SEND_SMS`, we hook into the `sendText()` and `sendMultipartText()` functions in the `SmsManager`. For intercepting usage of `RECEIVE_SMS`, we modify the callback function in the `ActivityManagerService`, which is called when an `IntentReceiver` finishes processing a broadcast intent. There is a parameter in this callback function for indicating whether or not the `IntentReceiver` wants to stop broadcasting a given intent. By checking this parameter, the monitor can detect the abortion of intent that notifies the receiving of a new SMS. For intercepting usage of the `WRITE_SMS`, we modify the `insert()` function of SMS content provider `SmsProvider`.

Note that a centralized permission check point is available in Android. However, simply hooking into this check point cannot fulfill the detection of permission abuses. This is because the captured information does not include what is needed for our purpose, such as how a permission is used (e.g., destination in the `SEND_SMS`) and behavior of the `abortBroadcast()`. Whenever a permission usage is captured, an appropriate method in the permission-abuse detector is called to trigger the verification procedure, during which the execution is suspended until a decision is made.

3.2 State Information Monitor

The main function of this component is to collect various state information related to each permission usage, including phone state, app state, and user input state. The phone state information incudes screen state, lock state and telephony state. We register an `IntentReceiver` to monitor the `ACTION_SCREEN_OFF` and `ACTION_SCREEN_ON` intents, and query the `KeyguardManager` to retrieve the state of the screen locker. In order to get notification immediately after a new SMS message is received, we modify the `SMSDispatcher` to report a new SMS message before the intent is even broadcasted. Finally, the `PhoneStateListenner` can help us keep tracking of the call state.

The app state information includes app visibility and app uptime. We trace the state of each app by tracking their `activities` that correspond to the same UID. With some slight modification to the `ActivityManagerService`, we can monitor the life-cycle transition of each `activity` (e.g., resume, pause, destroy). Then, we update the current user visible `activity` and its uptime accordingly. The longest activity uptime is the uptime of the target application. In addition, the `activity` information is also used to recognize the purpose-specific user input.

The collection of user input state information is challenging to implement. Since all user inputs are eventually sent to the application widget `EditText`, one may propose to capture user inputs by monitoring the text of each `EditText`. This method is, however, flawed because the `EditText` is actually controlled by the subject app, meaning that malicious apps can set or manipulate the text without being noticed by the user. In order to capture the text correctly, we need to make sure that we update the text of each `EditText` only when the user edits it. Our implementation achieves this by taking advantage of the Android Input Method Framework (IMF).

Recall that IMF consists of three components: `InputMethodManagerService` (IMMS), `Input Method` (IME), and client app. IMMS is a system service that manages the interaction between the IME and the client apps by binding the input view of the client app to the current IME. As a result, the client app can exchange the remote reference of the `InputConnection` class instance with the remote reference of the `InputMethodSession` class instance via IMMS. Then, the interaction between IME and the client app is accomplished through a remote procedure call (rpc) on the two remote references. For example, IME can commit an input text to the client app by calling the `commitText()` in the `InputConnection`, and the client app can update the cursor of the target view by calling the `updateSelection()` in the `InputMethodSession`.

In our prototype implementation, we modify the base class to duplicate the interaction between the IME and the client app to IMMS. In other words, IMMS will be notified whenever IME sends/receives a rpc request to/from the client app. The modified IMMS will update the current input text of the target view by calling the `getTextBeforeCursor()`, `getTextAfterCursor()` and `getSelectedText()` functions in the `InputConnection`. As a result, we can capture and temporarily store a user's input in IMMS. Because the update can be triggered by a user's editing activity only, we assure that the text is not faked by a malicious program that attempts to bypass DroidPAD. To prevent the leakage of sensitive user input information, we only allow queries from a system process.

Additional care must be taken as well. According to our system design, we not only need to provide the current user input information when permission verification is triggered, but also need to save the purpose-specific user input. Examples include: user input generated while composing a scheduled SMS message; defining the SMS filtering rules. To accomplish this, we keep tracking the names of `activity` in SMS apps, and use the names to perform the following tasks: composing scheduled and auto-reply SMS, inputting destination of auto-forward SMS, and configuring SMS blocking policy. With such information, we can automatically start caching the purpose-specific user input when we encounter these activities. The captured information will be stored in the configuration database for later use. The activity names are manually captured from all known SMS apps. For new apps that are released after installing DroidPAD, we can make DroidPAD automatically update its configuration database.

3.3 Permission-Abuse Detector

This component is the core of DroidPAD. We implement it as a system service because it needs to have the privilege for querying user inputs. It works as follows: Upon creation, it initializes itself according to the configuration database and starts tracking various state information (except for the user input) by utilizing the monitor. Once a verification process is triggered, it queries IMMS for user input with respect to the app. Together with other information that is collected by the state information monitor, a decision is made with respect to each permission usage as follows.

Detecting abuse of `SEND_SMS`.

We detect abuses of the `SEND_SMS` permission by mapping the intercepted permission usages to the pre-defined permis-

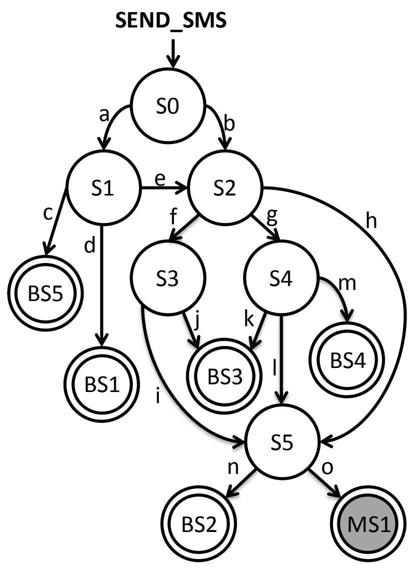

Figure 2: State machine for detecting abuse of `SEND_SMS`.

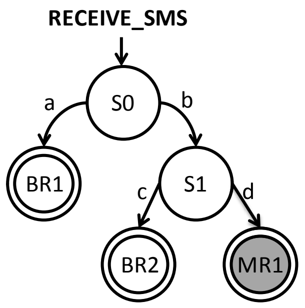

Figure 3: State machine for detecting abuse of `RECEIVE_SMS`.

sion usage scenarios. Figure 2 describes the implementation of the detector as a state machine. Specifically, the captured usage of `SEND_SMS` triggers a verification process and brings the detector into state *S0*. From *S0*, we check the pre-condition of the normal SMS sending activity (*BS1*), which includes screen state, lock state, app visibility and app uptime. Legitimate requests will transfer the detector state to *S1* (arc *a*), where the detector compares the sending message to the current user input. This leads to three possible results. (i) *Verified*: the message body is seen in the current user input and the detector reaches *BS1* (arc *d*). (ii) *Signature appended*: the SMS message body matches a user input with the appended text at the end and the detector reaches *BS5* (arc *c*). If benign scenario *BS5* is detected for the first time, it will be reported to the user so that user can approve the signature content for the next time. (iii) *Unverified*: There is no match with any current user input. Both unverified request and those denied by pre-condition check will drive the detector into state *S2* (arcs *b* and *e*). Then, as an example, we consider the telephony state information. In the case there are no telephony state changes, the detector follows arc *h* directly to reach state *S5*. On the other hand, arc *f* is traversed when there was an incoming phone call recently, and arc *g* is traversed when there is a new SMS message received. Both states *S3* and *S4* can lead to the identification of *BS3* after the auto-reply SMS content is verified. In addition, it is possible that the detector goes from *S4* to *BS4*, which requires to checking the pre-defined forwarding phone number. If no conclusion is drawn from this point, the detector will finally be in state *S5*. There is a last chance to prove the innocence from *S5*, namely that the detector tries to match a record in scheduled SMS (*BS2*). If unsuccessful, the request will be detected as *MS1* and the permission-abuse will be blocked and reported to the human user.

Detecting abuse of `RECEIVE_SMS`.

Figure 3 describes the state machine corresponding to this detector component. The idea for detecting abuses of `RECEIVE_SMS` is the following. The detector first checks the SMS database to make sure that the intercepted new SMS message is appropriately handled. If the new SMS message is written to the SMS database, then the detector is in state *BR1* (arc *a*); otherwise (arc *b*), we use the user pre-defined blocking policy to determine whether the permission usage is legitimate or not. Specifically, if the blocked SMS message satisfies a blocking policy that is extracted from the user input, the permission usage is determined as *BR2* (arc *c*); otherwise, it is determined as malicious (arc *d*).

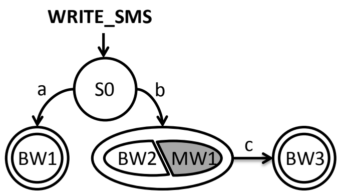

Figure 4: State machine for detecting abuse of `WRITE_SMS`.

Detecting abuse of `WRITE_SMS`.

Because the SMS content provider is a centralized database that is shared system-wide, any modification to it will have a global effect. With a single `WRITE_SMS` permission, malicious apps can forge an incoming SMS message as if it is sent by some entity (e.g., a company, bank, friend, or family). The inserted SMS message will then be automatically captured and displayed by other legitimate SMS apps. This detector component defeats the attack by making the unverified modification private to each app and by associating the database entry with its UID. All verified entries will be marked as 0 in the UID entry, which makes the detection mechanism transparent to the legitimate apps. As a result, malicious apps have to convince (or trick) the user to check the inserted SMS message as a premise of the attack. This means that it is no longer necessary to modify the SMS database because the attacker can simply fake a SMS message in the user interface.

Figure 4 describes the state machine for detecting abuse of this permission. Specifically, the verification of a `WRITE_SMS` request operates as follows. When an app attempts to insert an incoming SMS into database, the content that matches a newly received SMS message is verified (arc *a*); otherwise, it will be unverified (arc *b*). The reason the detector does not deny unverified writing attempts is that there are legitimate apps that write unverified SMS message into the database (*BW2*). To accommodate such legitimate operations, we use the aforementioned mechanism to make chat messages visible only in the context of the subject app (e.g., `Go SMS`). On the other hand, the detector denies all updating requests on the *address, body* and *uid* columns of the *sms* table in the database. For a legitimate write request (*BW1*), the detector assures the integrity of the SMS message. Note that it is possible that the restored SMS messages (*BW3*) may be mistakenly isolated because they are essentially a series of unverified messages written in a short time window. We address this issue by delaying the report of the malicious write requests. For each detected malicious write request, we wait for a small time window to see if it belongs to a series of write request caused by *BW3*, such that we can report *BW3* and ask for the user's confirmation. The SMS messages will remain isolated until after the user explicitly approves them.

3.4 Manager & Configuration Database

DroidPAD Manager is implemented as an Android app that enables the interaction between the user and the DroidPAD. It notifies the user whenever a permission-abuse is detected, provides an interface for the user to configure exceptions, and reports a false-positive incident whenever an alert message is prompted. The configured exception is then store in configuration database. In order to secure the communication between the DroidPAD Manager and the other components (i.e., permission-abuse detector, configuration database), we elevate the privilege of this component by signing it with the platform key.

The configuration database is a content provider that stores the configured exceptions, assistant information (i.e., special activity names), and the captured purpose-specific user input. It is also signed with the platform key. By declaring the read/write permission of the provider at the signature level, it guarantees that the provider can only be accessed/modified by the code signed with the same key, namely the DroidPAD Manager. Therefore, both privacy and integrity of the provider is protected.

4. ANALYSIS AND EVALUATION

DroidPAD has good usability because it is almost transparent to the human users, while the involvement of human user is to prompt the user to approve/deny activities for sending messages. Now we discuss security of DroidPAD and report its effectiveness and performance evaluation results.

4.1 Security Analysis

We argue that DroidPAD is secure against the threats specified in the threat model specified in Section 2.2. First, data origin integrity is assured because (i) we directly collect information from system components which are well protected under our assumption. (ii) the evasive attacker can not take advantage of those system components to generate fake information without compromise their integrity. For example, it is impossible for malicious app to fake user input by exploiting any feature of IMF. Second, data integrity and secrecy is well protected because the strong isolation between those data and third party apps. On one hand, data flow at runtime is only visible from system process, hence isolated from unprivileged apps. On the other hand, the content provider holding our data in the disk is signed with the platform key, which protects it from unauthorized access and modification. Finally, code integrity is assured by setting system partition as read-only, which is the default security enforcement in Android. From the above argument we conclude that DroidPAD achieves all the desired security objectives.

4.2 Effectiveness Evaluation

We now evaluate the effectiveness of DroidPAD because it is essentially based on machine learning algorithms, whereas the above security analysis assures that the inputs to its decision-making algorithms are authentic. To evaluate its effectiveness, we tested DroidPAD against 8 popular (benign) SMS apps (i.e., `Stock SMS`, `Go SMS`, `Handcent`, `Chomp SMS`, `Youni SMS`, `AutoSm`, `SMS scheduler`, `SMS Blocker`) and the 40 malicious apps described in Table. 2, which includes 39 malicious apps from the Android Malware Genome Project [13] and 1 synthetic malware sample. The metrics are false-positive rate and false-negative rate. Evaluation experiments are performed with individual permissions.

Table 2: Malicious apps dataset

Permissions	Malware Samples
`SEND_SMS`(27)	FakePlayer(6), Walkinwat(1), HippoSMS(4), JiFake(1), Zsone(12), OpFake(3)
`RECEIVE_SMS`(12)	RogueLemon(2), Zitmo(1), RogueSPPush(9)
`WRITE_SMS`(1)	Syn1[‡](1)

[‡] Malicious app we developed for test purpose.

Effectiveness in detecting `SEND_SMS` *abuses.*

In our experiments, for each app we simulated all of the usage scenarios mentioned above and sent out 10 SMS messages in each scenario. The number of false alerts are summarized in Table 3. The false alerts corresponding to the `Go SMS` app is originated from appended signature *BS5*. Specifically, we got a false alert at the first attempt and approved the detected signature for the rest of SMS messages. We also got two false alerts from the `Youni SMS` app while sending an auto-filled invitation SMS message to friends in the contact book. Because no user input occurred in this process, DroidPAD identified them as background SMS messages. Except for the three false alerts, DroidPAD successfully verified the other 137 SMS messages. In other words, the false-positive rate is 2.1%.

In order to measure false-negative rates, we tested DroidPAD against 27 malicious apps (Android malwares from 6 families), all of which send background SMS messages without being noticed by the human user. Experiment results show that all these abuses of `SEND_SMS` are detected and blocked by DroidPAD in real time, with a zero false-negative rate.

Effectiveness in detecting `RECEIVE_SMS` *abuses.*

The benign apps in the evaluation experiment consists of the first five apps in Table 3, which declared the `RECEIVE_SMS` permission. We also experiment with another publicly available app called `SMS Blocker`. All these apps, except `Stock SMS`, have the feature of blocking unwanted SMS messages based on user-configured blacklists. Because

Table 3: False alerts reported in the experiment for detecting SEND_SMS abuses

	Normal SMS	Scheduled SMS	Auto-replied SMS	Auto-forwarded SMS
Stock SMS	0	-	-	-
Go SMS	0 | 1^*	0	0	-
Handcent	0	0	-	-
Chomp SMS	0	-	-	-
Youni SMS	2	-	-	-
AutoSms	0	0	0	0
SMS scheduler	-	0	-	-

the RECEIVE_SMS function of different apps may conflict with each other, we conducted the experiment with one app at a time. The blacklists are configured in different ways with the following inputs: the phone numbers selected from the contact book or the call history or the SMS log (marked as †), the phone numbers that are manually typed in (marked as ‡), the keywords that are manually typed in (marked as §), and the prefix or suffix of phone numbers that are manually typed in (marked as ◇). For each app, we configure one blocking rule for each of the aforementioned 4 configuration types. Then we send 10 SMS messages to the emulator with the unwanted messages, one per configured rule. As shown in Table 4, the total unwanted SMS messages account for 20% (12 out of 60) of the total received SMS messages, which would be much smaller in real-life settings. Because all unblocked SMS messages are written into the SMS database, we encounter zero false-alerts on legitimate SMS messages (*BR1*). All false-alerts are generated by unwanted SMS messages that are blacklisted via phone numbers that are selected from a list (e.g., contact, call log), which means that no user input can be captured by DroidPAD (one type of *BR2*). Overall, the false-positive rate is 8.3% (5 out of 60).

Table 4: False alerts reported in the experiment for detecting RECEIVE_SMS abuses, where $a(b)$ indicates a false alerts out of b received SMS messages.

	Good SMS	Unwanted SMS
Stock SMS	0 (10)	-
Go SMS	0 (6)	1 ($1^\dagger$,$1^\ddagger$,$1^\S$,$1^\diamond$)
Handcent SMS	0 (8)	1 ($1^\dagger$,$1^\ddagger$)
Chomp SMS	0 (9)	1 ($1^\dagger$)
Youni SMS	0 (9)	1 ($1^\dagger$)
SMS Blocker	0 (6)	1 ($1^\dagger$,$1^\ddagger$,$1^\S$,$1^\diamond$)

We further experiment with 12 real malwares from 3 families (i.e., RogueLemon (2), RogueSPPush (9), Zitmo (1)) that block incoming SMS messages from certain number (e.g., 10086) silently. Again, DroidPAD detected all the permission abuses. In total, DroidPAD missed 9 abuses of the RECEIVE_SMS permission (*MR1*).

Effectiveness in detecting WRITE_SMS *abuses.*

We use the first 5 SMS apps in Table 3 to test the effectiveness of detecting WRITE_SMS abuses. Four out of the five apps allow the user to send and receive free (or cheaper) SMS messages via their own services, instead of the service that is provided by the carrier (*BW2*). Our experiment results show that DroidPAD successfully differentiates the free SMS messages from the legitimate SMS messages. In total, we encounter 0 false alerts. We also test *BW3* by restoring the SMS messages from a previously created backups. As expected, DroidPAD isolates all the restored SMS messages and presents the user with an alert message. This can be categorized as false positive as we identify this special benign usage scenario as malicious. However, a confirmation from user can help us act further on it, which will cancel the isolation enforced by DroidPAD. We argue that this requirement on user will not undermine the overall user experience as *BW3* is relatively a rare case.

We further test a simple Android malware, which is developed to fake a SMS message that has an embedded URL pointing to another malicious URL (*MW1*). This malware can successfully launch the smishing attack on a stock Android image. This attack is also detected and blocked by DroidPAD in real time.

In summary, DroidPAD if very effective as the (false positive rate, false negative rate) in detecting abuses of SEND_SMS is (2.1%, 0%), in detecting abuses of RECEIVE_SMS is (8.3%, 0%), and in detecting abuses of WRITE_SMS is (0%, 0%).

4.3 Performance Evaluation

For performance, we test DroidPAD with respect to a set of standard benchmarks. The experiments are performed on an emulator running Android version 4.12, which is modified to incorporate DroidPAD as mentioned above.

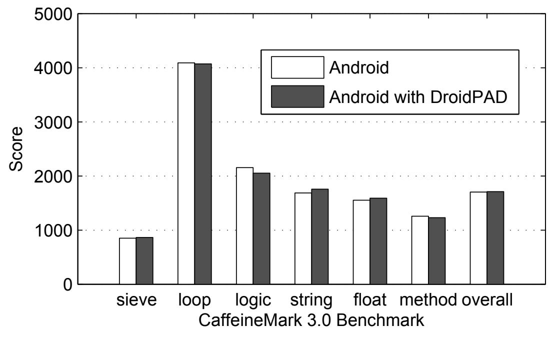

Figure 5: Performance evaluation results.

The benchmark results are shown in Figure 5. We observe that the performance difference between the two systems is almost negligible, meaning that DroidPAD does not incur any significant performance overhead. We also observe that sometimes DroidPAD actually leads to slightly better performance. This might be caused by other factors that are not related to our system.

5. RELATED WORK

The recently released Android 4.2 started requesting users' confirmation whenever a SMS message is to be sent to a premium number. This approach is based on the idea of blacklisting and is therefore different from ours. While it is ideal not to modify the Android system (e.g., enforcing policy via repackaging [11]), most solutions (including our own) require to modifying it somewhat as we elaborate below.

There are studies on extending the Android permission system to enforce fine-grained control *at runtime*. Examples are: allowing a user to authorize partial permissions and revoke granted permissions on demand [8]; enforcing usage control based on system context (e.g., device location) [1]; offering a privacy mode to enable dynamic management of privacy-related permissions [15]. These studies require the users to have a good understanding of permissions as well as policies. In contrast, we aim to minimize, or at least mitigate, the burden imposed on the users.

There are studies on enhancing *install-time* permission assignment. Example are: detecting undesired combination of requested permissions [4]; mapping Android API to permission labels and detecting permission overprivilege [5]. However, malicious apps may request permissions that may not trigger any of such detection systems. It is also possible to enforce control on both *install-time* permission assignment and *runtime* permission usage [9], which however may impose the app developer to specify policies. In contrast, we aim to automatically detect and block (prevent) permission abuses in real time.

There are studies on *offline* characterization of Anroid attacks. Examples are: tracking information flow to detect data stealing apps [3]; detecting malicious apps by system call based clustering [2]; discerning malicious apps in the market based on malicious behavior footprints [14]; evaluating the potential risk of given apps according to its behaviors [7]; analyzing malware using virtualization technology [12]. In contrast, our approach aims to automatically detect and block (prevent) permission abuses in *real time*.

6. FUTURE WORK AND CHALLENGES

The current proof-of-concept prototype implementation focuses on detecting abuses of three SMS permissions. However, the current implementation does not capture direct voice input for composing SMS messages, because voice input does not go through IMF. One can extend DroidPAD to support voice input by modifying the voice recognition service, whose source is unfortunately not available in AOSP at this moment.

As mentioned above, we aim to design and implement a system that can, ideally, detect and block all permission abuses in real time. We foresee some challenges in the course of implementing the envisioned full-fledged system. In particular, the current implementation of DroidPAD is based on manually profiling of some benign and malicious apps. When we extend the study to deal with all permissions, we encounter a large number of apps. This means that we have to automate the profiling process. This in principle can be achieved via Machine Learning, but we must adequately address their false-positives and false-negatives.

7. CONCLUSION

We have motivated the need of a system for detecting and blocking permission abuses in real time. As a first step towards this ultimate goal, we presented the design, implementation and analysis of DroidPAD that aims to detect and block abuses of three SMS-related permissions in real time. Experimental results showed that DroidPAD incurs very low false-positive and false-negative rates, as well as an unnoticeable performance overhead. We also discussed challenges that must be addressed to achieve the ultimate goal.

Acknowledgement. We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. This work was supported, in part, by AFOSR under Contract FA9550-09-1-0165 and NSF under grants 1111925, 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFOSR or NSF.

8. REFERENCES

[1] Bai, G., Gu, L., Feng, T., Guo, Y., and Chen, X. Context-aware usage control for android. In *SecureComm* (2010), Springer, pp. 326–343.

[2] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. Crowdroid: behavior-based malware detection system for android. SPSM '11, ACM, pp. 15–26.

[3] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. OSDI'10, USENIX Association, pp. 1–6.

[4] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. CCS '09, ACM, pp. 235–245.

[5] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. CCS '11, ACM, pp. 627–638.

[6] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions: user attention, comprehension, and behavior. SOUPS '12, ACM, pp. 3:1–3:14.

[7] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: scalable and accurate zero-day android malware detection. MobiSys '12, ACM, pp. 281–294.

[8] Nauman, M., Khan, S., and Zhang, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS* (2010), ACM, pp. 328–332.

[9] Ongtang, M., McLaughlin, S. E., Enck, W., and McDaniel, P. D. Semantically rich application-centric security in android. In *ACSAC'09*, IEEE Computer Society, pp. 340–349.

[10] Wikipedia. Smishing — Wikipedia, the free encyclopedia, 2013. [Online; accessed 30-Jan-2013].

[11] Xu, R., Saïdi, H., and Anderson, R. Aurasium: practical policy enforcement for android applications. Security'12, USENIX Association, pp. 27–27.

[12] Yan, L. K., and Yin, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. Security'12, USENIX Association, pp. 29–29.

[13] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy* (2012), IEEE Computer Society, pp. 95–109.

[14] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. NDSS'12.

[15] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. W. Taming information-stealing smartphone applications (on android). In *TRUST* (2011), Springer, pp. 93–107.