Towards a Systematic Study of the Covert Channel Attacks in Smartphones

Swarup Chandra[1](✉), Zhiqiang Lin[1], Ashish Kundu[2], and Latifur Khan[1]

[1] The University of Texas at Dallas, Richardson, TX, USA
{swarup.chandra,zhiqiang.lin,lkhan}@utdallas.edu
[2] IBM T J Watson Research Center, Yorktown Heights, NY, USA
akundu@us.ibm.com

Abstract. Recently, there is a great attention on the smartphones security and privacy due to their increasing number of users and wide range of apps. Mobile operating systems such as Android, provide mechanisms for data protection by restricting the communication between apps within the device. However, malicious apps can still overcome such restrictions via various means such as exploiting the software vulnerability in systems or using covert channels for data transferring. In this paper, we aim to systematically analyze various resources available on Android for the possible use of covert channels between two malicious apps. From our systematized analysis, we identify two new hardware resources, namely battery and phone call, that can also be used as covert channels. We also find new features to enrich the existing approaches for better covert channel such as using the audio volume and screen brightness. Our experimental results show that high throughput data transmission can be achieved using these resources for the covert channel attacks.

Keywords: Android · Covert Channel · Mobile Security

1 Introduction

Smartphone users today install multiple apps that provide personalized services and easy access of users' personal information including credit card, medical records, phone contacts, insurance card, etc. Data security of these sensitive information has become a critical concern to these users. Android operating system (OS) inherits the Linux security infrastructure where apps are installed and executed within its individual virtual environment or sandbox [4]. The OS uses security policy based permission model to control the access to the shared resources, and an app has to seek the explicit permissions to access them during the installation time.

An attacker interested in obtaining user's private data must circumvent the security policies that prevent the illegal access. In Android, covert channels can be used by malicious apps for such an attack. A covert channel is a medium through which two entities communicate without using conventional methods

J. Tian et al. (Eds.): SecureComm 2014, Part I, LNICST 152, pp. 427–435, 2015.
DOI: 10.1007/978-3-319-23829-6_29

(e.g., intents). In particular, an app having access to user's private data can transfer it to another app within the same device, or to an external server using these non-conventional channels. This data transfer can be oblivious to an end user. It has been generally accepted that a covert channel of bandwidth>100 bps would pose a significant threat to data security in a system [15]. Therefore, the existence of large bandwidth covert channels pose a high risk of storing user's private data on a mobile device.

Since shared resources are typically used as a medium for covert channel communication between two entities. Shared resource attributes which provide apps the ability to read, store and modify data, can be exploited by malicious apps to execute a communication protocol for data transfer. As such, it is imperative to identify these possible communications and mitigate their threats. The threat model considered in this paper involves a malicious app (*App A* or Encoder) having access to user's private data, transfers the information to another app (*App B* or Decoder) on the same device which does not have access to this data, using a covert channel.

In this paper, we systematically analyze the properties of various shared resources on an Android system and evaluate their use as possible covert channels for establishing communications between two malicious apps installed on the same device. Specifically, by using a shared resource matrix approach [12] to inspect each shared resource attribute that satisfies covert channel properties, we discover new storage and timing covert channels, which have not been studied before. In particular, we show that the use of battery and phone call frequency as timing channels and the use of phone call log as storage channels are realistic threats. In addition, we enumerate other shared resources shown in existing studies such as audio and screen to find new features and observations which can be used to develop a better covert channel. Our experimental results show that these channels can have sufficiently high throughput and cannot be ignored.

2 Background and Related Work

Covert channels can be classified as timing or storage channels. In a timing channel, information between two colluding apps is transmitted using shared resources having no storage capability (e.g., CPU) for a specific period of time. The encoding and decoding of information is performed with precise time synchronization between the two apps. In contrast, a storage channel involves the use of shared resources having storage capability (e.g., file system). This enables asynchronous encoding and decoding of the information.

Identification of covert channel on a system is known to be a hard problem [13]. There have been multiple studies that identify various shared resources supporting covert communication in Android such as the network channel [5]. A recent study involves the design of a real world malware app that uses audio and system settings as covert channels [17]. A survey of various covert channels [14] demonstrate possible timing and storage channels. However, none of the existing work has performed a systematic study on all shared resources and their properties in an Android system.

Table 1. Overview of the shared components between apps, and the possible use of covert channel attack. (Symbol ✔ denotes a covert channel is possible and have been studied, ✗ the covert channel attack has not been studied yet)

Application Level											
System Settings	✔	[6, 14]	Intents	✔	[14]	System Services	✔	[17]	Content Providers	✔	[16]
OS Level											
Sockets	✔	[14, 16]	/proc/	✔	[14]	File System	✔	[1, 14]	System Log	✔	[14, 16]
Hardware Level											
CPU	✔	[14]	Sensors	✔	[14, 17]	Battery	✗	-	Screen	✔	[14, 17]
Memory	✔	[14, 16]	Vibrator	✔	[14, 19]	Audio	✔	[17]	Phone	✗	-
Camera	✔	[18]	Bluetooth	✔	[14]	USB	✗	-	Network	✔	[5]

In order to perform a systematized analysis of shared resource properties that can support covert channels, we identify attributes that satisfy covert channel properties [2] of each possible resources that are enumerated in Table 1. These resources are classified as application level, OS level and hardware level based on their attribute properties [14]. Previous studies have already identified many shared resources that could form a covert channel, which is also summarized in the table. We can observe that the hardware such as Battery and Phone component have not been studied before to form a covert channel (indicated by ✗), and the goal of this work is to find them out and demonstrate their feasibilities.

3 Analysis Overview

In this section, we describe how we identify the shared resources (using a shared resource matrix [12]), and inspect each of them to form a covert channel between colluding apps within a device. In general, a shared resource that can be used to form a covert channel needs to satisfy at least the following capability:

- Ability for apps to *read* from a resource.
- Ability for apps to *write* to a resource.
- Ability for apps to turn *on* (or *off*) a resource.
- Ability for an app to *lock* a resource, in which case other apps cannot access the same resource simultaneously.

To enumerate each shared hardware resource and check their properties, we have created a shared resource matrix shown in Table 2. Note that hardware resources are inherently shared by various apps in a device (because of multiplexing). For each resource that have the above capabilities (indicated by a ✔ on both *read* and *write*, *lock*, or *on/off* property), we check whether we can form a covert channel based on the following properties:

- Both the sending and receiving processes must have access to the same attribute of a shared object.
- The sending process must be able to modify the attribute of a shared object in the case of storage channel, or they must have access to a time reference, such as a real-time clock, a timer or the ordering of events in the case of a timing channel.
- The receiving process must be able to reference that attribute of the shared object.
- The sending process must be able to control the detection time by the receiving process, for a change in attribute value.
- A mechanism for initiating both processes, and properly sequencing their respective accesses to the shared resources, must exist.

Table 2. A Shared Resource Matrix on Android. (Symbol ✔ denotes satisfying the covert channel property; ✗ indicates not; ◐ denotes the covert channel attack we enriched, and ● denotes the brand new covert channel we identified.)

		Shared Hardware Resources											
		CPU	Sensors	Vibrator	Battery	Screen	Memory	Audio	Phone	Camera	Bluetooth	Network	USB
Property	Read	✔	✔	✗	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Write	✔	✗	✔	✔	✔	✔	✔	✔	✗	✗	✗	✗
	Lock	✗	✗	✗	✗	✗	✔	✗	✔	✔	✔	✗	✗
	On/Off	✗	✔	✔	✗	✔	✗	✔	✗	✗	✔	✔	✗
Covert Channel?		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✗
Brand New?		○	○	○	●	◐	○	◐	●	○	○	○	-

As shown in Table 1, we find 11 out of 12 resources listed that satisfy covert channel properties, and these resources include such as Battery, Screen, Audio and Phone. Among them, resources such as CPU, Memory, Sensors, Vibrator, Camera, Bluetooth and Network have already been identified in earlier work. Hence, we do not consider them for our analysis. Interestingly, we do find two new covert channels (denoted by ●), namely Battery and Phone Calls (details in Sect. 4). Also, for the two previously studied covert channels (denoted by ◐), we enrich them by using attributes not specified in these studies (details in Sect. 5).

4 Discovery of New Covert Channels

With our systematized analysis, we have identified two new covert channels: Battery and Phone Call. In this section, we present the details of our discovery.

Battery. Mobile devices typically have a Lithium-ion battery, with limited charge capacity. Parallel use of multiple resources discharges the battery at different rates depending on the component used. This property can be exploited for encoding of information to form a covert channel. Specifically, the `Battery Manager` API provides a broadcast intent [8] informing an app (with intent filter registered with `ACTION_BATTERY_CHANGED`) about every 1 % change in the battery charge level. A malicious app can perform a binary encoding of desired information by running parallel operations on combination of resources such as CPU and screen brightness, to achieve a predetermined discharge rate. A decoder estimates the discharge rate for an exact time period using the broadcast intent, thereby forming a covert channel.

Phone Call. Apps with `CALL_PHONE` permission can make a phone call using an intent with `ACTION_CALL`, and end the call using a Java reflection method involving the `ITelephony` interface. This ability to make phone calls can also be exploited to form covert channels. More specifically, there could be two such channels.

- *Phone Call Frequency Channel*: Apps can place phone calls at a predetermined frequency to encode binary values. Colluding app having `READ_PHONE_STATE` permission can synchronously measure the call frequency by registering a receiver to a broadcast intent from `TelephonyManager` API informing of a change in call state [11]. Since both colluding apps require exact time synchronization, this is a timing channel.
- *Phone Call Log Channel*: Apps can dial an integer value encoding a desired information in the `URI` attribute of the phone call intent. The dialed number is stored in a call log content provider, which can be read by a decoder with `READ_CALL_LOG` permission. The information stored is determined by checking the latest dialed number from the call log [9]. Since the dialed number is stored in the call log as an ASCII string, the length of the number can be arbitrarily large. The two colluding apps do not require exact time synchronization since this is a storage channel.

5 Enrichment of Existing Covert Channels

We reported in Table 2 that there are also existing efforts (e.g.,[6,14,17]) on using resources such as screen and audio for covert channels. While existing work did show their feasibility, in this section we would like to concretize and enrich them on how we would like to exploit them in the covert channel attacks.

Screen. Screen resource attribute such as system settings can be set by apps as shown in [14,17]. Here, we analyze a specific attribute namely `SCREEN_BRIGHTNESS` system settings parameter, which is not specifically mentioned in early studies. Screen brightness can be changed to an appropriate integer value in the range of 0 to 255 by an app accessing system settings [10], if the `SCREEN_BRIGHTNESS_MODE` parameter is set to 0. A decoder can read the encoded integer value which may represent a desired information.

Audio (Volume). Prior efforts (e.g., [14,17]) identified the audio channel using system settings and APIs involving the volume attribute. Here, we provide new insights regarding the use of multiple API components forming a volume based covert channel. The `AudioManager` API provides multiple stream volume components including `STREAM_ALARM`, `STREAM_DTMF`, `STREAM_MUSIC`, `STREAM_NOTIFICATION`, `STREAM_RING`, `STREAM_SYSTEM` and `STREAM_VOICE_CALL` [7]. Similar to system settings, apps can set integer values on each component representing a volume level, using `setStreamVolume` method. This property can be exploited for encoding desired information using a combination of volume components. A range of integer values allowed for each component can be obtained using the `getStreamMaxVolume` method.

6 A Covert Channel Protocol

In this section, we briefly describe the design and implementation of a communication protocol we developed to enable covert communication between two malicious apps using channels in Sects. 4 and 5. More details about this protocol can be found in our technical report [3].

Major challenges in using shared resources as covert channels include noise due to external factors such as parallel app execution or end user interaction, scheduling uncertainty, and bandwidth limitation. We can overcome these challenges by designing a synchronization protocol [17] to enable sequential ordering of data transfer events between the two colluding apps. In particular, noise due to uncertainty in scheduling of encoding and decoding operations occurs due to parallel execution of the two colluding apps. Synchronization of these parallel processes can be performed by a clocking mechanism that schedules execution of one operation at a time from the encoder or the decoder, thereby reducing the noise. Further, limitations due to external factors can be overcome to a certain extent by simple checks for protocol disruptions such as unexpected change in channel value. Finally, bandwidth limitation can be addressed by splitting the desired information into binary strings of appropriate length. For example, an integer value ≤ 255 corresponds to a binary string of length 8 bits. This integer can be encoded using the Screen channel (called a data channel) whose supported range is 0 to 255. If the desired data (binary string) is of length greater than 8 bits, the data is split into multiple chunks, each of length 8 bits. These chunks are then transmitted over the screen channel sequentially by converting the binary value into a corresponding integer.

The two colluding apps initialize using a single bit channel (also called sync bit) to begin data transfer. In case of a storage channel, encoder sets the sync bit to 1 after encoding a data chunk on a channel, and waits for a response from the decoder before encoding the next chunk. The decoder responds by flips this sync bit to 0 after successfully reading the data channel. Conversely, the sync bit indicates start and end of encoding in a timing channel, which is used by the decoder to exactly synchronize with the encoder.

Implementation of covert timing channels requires the evaluation of different thresholds representing 0 and 1. We empirically determine the thresholds for Phone Call Frequency channel (number of calls that can be placed per second) and Battery channel (amount of battery discharge percent is achieved by parallel use of different components) using our test phone [3]. In the case of Battery channel, we performed a parallel execution including CPU, Screen Brightness, Cellular network, Vibrator, GPS data, and Phone component to determine the threshold values. More details on how we get these threshold values can be found in our technical report.

7 Evaluation and Discussion

We now present our experimental results on the covert channels we have analyzed. Evaluation of each channel was performed on a Samsung Galaxy S phone running Android version 4.2.2.

Table 3. Protocol statistics with *Throughput*: Ratio of Input Length and Time Taken

Covert Channel		**Supported Range**		**Input**	**Time**	**Throughput**
		Integer Range	**Binary Length**	**Length L (bits)**	**Taken T (sec)**	**L/T (bps)**
Phone Call Log		-	2.3M (max)	2.3M (max)	67.3	34175.3
Phone Call Frequency		0 - 1	1	10	16.05	0.623
Screen		0 - 255	8	525	0.828	634.05
Audio (Volume)	DTMF (D) Music (M) Alarm (A) Notification (N)	D (0 - 15) M (0 - 15) A (0 - 7) N (0 - 7)	D = 4 M = 4 A = 3 N = 3 Total = 14	525	1.6	328.125
Battery		0 - 1	1	5	1515.15	0.0033

Experiments involve data transfer of a random binary string of certain length (given under *Input Length* column in Table 3), from an encoder to a decoder using each covert channel mentioned in Sects. 4 and 5. As mentioned in Sect. 6, the binary string is divided into data chucks of appropriate size (given under *Binary Length* column in Table 3) for each channel.

The table shows the throughput obtained in our experiments on each channel, averaged over 10 experiments with different randomly selected input binary string. We performed various experiments using the Phone Call Log channel with multiple input lengths. We observed a near-linear increase in transfer time with exponential increase in input length for this channel (more details are presented in [3]). Therefore, the highest throughput of 34.17 kbps ($= \frac{2.3M}{67.3}$) was obtained by transferring 2.3M bits in 67.3 secs after encoder and decoder initialization. However, we observed a decrease in responsiveness of answering a query to the call log content provider with increase in input length. This negatively affected the throughput beyond the input length of 2.3M bits on our test phone. Such a behavior may be due to memory limitations of the call log content provider query mechanism. Further, we obtain a higher throughput on the Screen and Volume channel than previously reported in [14] and [6] respectively. This is primary due to the use of higher bandwidth attribute(s) to form the channels. Additionally, the table shows a lower throughput on the Volume channel which uses 14 bits, compared to the Screen channel which uses only 8 bits. This is due to slower response time of `AudioManager` API, and larger time required to set and read multiple attributes in the Volume channel as compared to a single attribute in the Screen channel.

On the other hand, a low throughput obtained using the Phone Call Frequency channel can also be attributed to low bandwidth, and intent scheduling uncertainty. Phone calls are placed using an intent which contains the number dialed, as explained in Sect. 4. During the experiments, we found that these intents are not scheduled at a desired frequency by the intent handler. This may be due to interference from multiple process calls generated for handling each intent. Finally, in case of the Battery channel, a faster battery discharge is required to obtain higher throughput. However, the table shows an extremely low throughput. Our empirical threshold estimation considered a bandwidth margin beyond the average battery discharge rate due to normal device operation. On an average, it took at least 5 mins to achieve such a discharge rate for encoding a single bit. This can be attributed to the difficulty of an app to drain the battery using different resources on a device since these resources are typically designed to consume minimal power.

One possible way to reduce the bandwidth of the Phone Call Log channel is to limit the string length of each record stored in the call log. A limitation of the channel over the phone component is that its usage cannot be made oblivious to an end user. The user can easily detect a phone call being made, or review the phone call log for dialed numbers. We leave the evaluation of channel obfuscation to avoid detection, for future work.

8 Conclusion

We have presented a systematic study of the shared resources available to an app on an Android phone and evaluated their possibility of support of a covert channel. In particular, we analyze various shared hardware resources that can be potentially exploited to transfer data maliciously between two apps on the same device. Our analysis yields two novel types of covert channel attacks that involves the battery and the phone component. We also design a communication protocol that can be used to achieve high throughput among the shared resources we inspected, and overcome the limitations in data transmission by using a synchronization mechanism between two colluding apps. Our study shows that a high throughput, greater than 30kbps, can be achieved with the use of phone component as a covert channel.

Acknowledgment. We thank anonymous reviewers for their invaluable feedback. This research was partially supported by The Air Force Office of Scientific Research under Award No. FA-9550-12-1-0077. Any opinions, findings and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

References

1. Ali, M., Humayun A., Zahid, A.: Enhancing stealthiness & efficiency of android trojans and defense possibilities (EnSEAD)-android's malware attack, stealthiness and defense: an improvement. In: Frontiers of Information Technology (FIT). IEEE (2011)
2. Bishop, M.: Introduction to computer security. Addison-Wesley Professional, Amsterdam (2004)
3. Chandra, S., Lin, Z., Kundu, A., Khan, L.: Towards a Systematic Study of the Covert Channel Attacks in Smartphones. Technical report, University of Texas at Dallas (2014)
4. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: USENIX Security Symposium, vol. 2, p. 2, August 2011
5. Gasior, W., Li Y.: Network covert channels on the Android platform. In: Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research. ACM (2011)
6. Hansen, M., Raquel, H., Seth, W.: Detecting covert communication on Android. In: 37th Conference on Local Computer Networks (LCN). IEEE (2012)
7. http://developer.android.com/reference/android/media/AudioManager.html
8. http://developer.android.com/reference/android/os/BatteryManager.html
9. http://developer.android.com/reference/android/provider/CallLog.Calls.html
10. http://developer.android.com/reference/android/provider/Settings.System.html
11. http://developer.android.com/reference/android/telephony/TelephonyManager.html
12. Kemmerer, R.A.: Shared resource matrix methodology: an approach to identifying storage and timing channels. ACM Trans. Comput. Syst. (TOCS) **1**(3), 256–277 (1983)
13. Lampson, B.W.: A note on the confinement problem. Commun. ACM **16**(10), 613–615 (1973)
14. Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S.: Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of the 28th ACSAC, pp. 51–60. ACM, December 2012
15. NCSC, NSA.: Covert Channel Analysis of Trusted Systems (Light Pink Book). NSA/NCSC-Rainbow Series publications (1993)
16. Ritzdorf, H.: Analyzing Covert Channels on Mobile Devices. Diss. Master thesis ETH Zrich (2012)
17. Schlegel, R., Zhang, K., Zhou, X. Y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: NDSS, vol. 11, pp. 17–33, February 2011
18. Simon, L., Ross A.: PIN skimmer: inferring PINs through the camera and microphone. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices. ACM (2013)
19. van Cuijk, W.P.M.: Enforcing a fine-grained network policy in Android (2011)