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

Abstract. Recently, there is a great attention on the security and privacy issues in smartphones due to their increasing number of users and wider 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 through various ways such as exploiting the software vulnerability in systems or using covert channels for data transfer. In this paper, we 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 be used in covert channels. We also find two new features in screen and audio, and enrich the existing approaches for better covert channel. In addition, we provide a detailed attack scheme using synchronization mechanisms between the two colluding apps for data transmission. Our experimental results show that high throughput data transmission can be achieved using these resources for covert channels.

Key words: Android, Covert Channel, Mobile Security

1 Introduction

Smartphone users today install multiple apps that provide personalized services on their device. These services allow easy accessibility and convenient storage for user's personal information including credit card, medical records, phone contacts, insurance card, etc. Data security of such sensitive information has become a critical concern for the end users. In most cases, the operating system provides necessary security infrastructure to protect user's data from malicious apps. Android is one such operating system that inherits the Linux security infrastructure. Apps are installed with different user ID's (unless explicitly specified by the app developer), and are executed within its individual virtual environment or sandbox [10]. The operating system uses security policy based permission model to provide specific access permissions to shared resources. Apps usually seek these permissions to access shared resources at installation time [2].

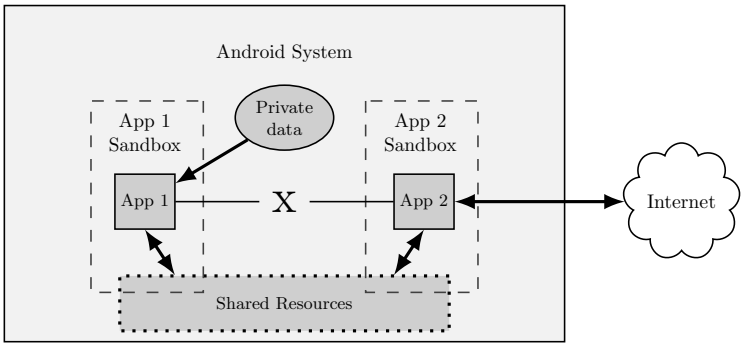

Fig. 1. Threat Model: An overview of a covert channel between two malicious apps.

An attacker interested in obtaining user's private data must circumvent security policies that prevent data access or communication between malicious entities. In Android, these attacks involving apps have been successfully demonstrated to have high bandwidth [10]. One such type of attack uses covert channels for data transfer (or exchange). A covert channel is a medium through which two entities communicate without using conventional methods (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. For example, a malicious app having access to user's medical records can transfer the data to a server via the internet by encoding the data in terms of network delays over legitimate traffic [13]. This data transfer can be oblivious to the end user since it is hard for the operating system to recognize the existence of such encoding medium or covert channels. It has been generally accepted that a covert channel of bandwidth >100 bps would pose a significant threat to data security in a system [20]. Therefore, the existence of large bandwidth covert channels pose a high risk of storing user's private data on a mobile device.

Covert channel typically involves the use of shared resources as a medium for communication between two malicious 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 covert data transfer. More specifically, Figure 1 shows the threat model considered in this report. It involves a data transfer between two malicious apps installed on the same device, using shared resources as covert channels. A malicious app (*App A* or *Encoder*) have appropriate privileges to access user's private data. However, in order to prevent illegal transfer of this data to an untrusted destination, *App A* does not have Internet access. This may enhance user's trust in the app to provide data security. Another malicious app (*App B* or *Decoder*) installed on the same device have Internet access, and is restricted from accessing the user's private data. It is also restricted from communicating with *App A* via normal communication channels (e.g., intent). These two apps are assumed to be developed by the same developer whose malicious intent is to obtain the user's private information via the Internet. This can be possible only by transferring data from *App A* to *App B*,

which can then transfer the data to the attacker via Internet. In this case, *App A* transfers the private data to *App B* via a covert channel, using a shared resource. As mentioned in earlier studies [19], such a scenario is possible since users can be easily lured to install these two malicious apps on their device. Therefore, it is imperative to find out these possible communications and mitigate them if we cannot stop them.

In this technical report, 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 [15] 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. We also 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. In addition, we present a communication protocol that uses multiple covert channels for a synchronized data transfer between colluding apps, reducing possible noise due to external interference. Our experimental results show that these channels can have sufficiently high throughput and cannot be ignored.

2 Background and Related Work

In this section, we give some background information on covert channels in Android, along with a survey of related work. Covert channels provide a medium for communication between colluding apps for malicious exchange of users' sensitive data, using shared resources to avoid being detected or restricted. This undesirable exchange of information invalidates the explicit or implicit communication restrictions between apps when using standard communication channels. Therefore, a systematic identification of resources that can potentially form a high bandwidth channel, is necessary to understand and mitigate this threat.

Covert channels can be broadly classified as *timing* or *storage* channel [16]. In Android, data encoding using a timing or a storage channel is performed by an app (encoder). Encoding using a timing channel is performed by utilizing or modifying a memoryless attribute of a shared resource (such as CPU), for a specific amount of time. A receiver (decoder) decodes data by checking changes on the channel attribute for this exact time period, according to a decoding protocol. This channel needs precise time synchronization between the two colluding apps since it does not store any encoded information. In contrast, data transmitted over a storage channel is stored using a shared resource attribute for a finite amount of time. The shared resource attribute typically have some form of memory which can store encoded information. This stored data can be read by a decoder accessing the same shared resource attribute asynchronously.

Application Level											
System Settings	✔	[14, 19]	Intents	✔	[19]	System Services	✔	[29]	Content Providers	✔	[22]
OS Level											
Sockets	✔	[19, 22]	/proc/	✔	[19]	File System	✔	[1, 19]	System Log	✔	[19, 22]
Hardware Level											
CPU	✔	[19]	Sensors	✔	[19, 29]	Battery	✗	-	Screen	✔	[19, 29]
Memory	✔	[19, 22]	Vibrator	✔	[19, 31]	Audio	✔	[29]	Phone	✗	-
Camera	✔	[30]	Bluetooth	✔	[19]	USB	✗	-	Network	✔	[13]

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, ✗ denotes the covert channel attack has not been studied yet)

Identification of all covert channels on a system is known to be a hard problem [17]. Multiple studies identify various shared resources that support covert communication in Android. This includes network channel where the app communicates with a remote server in an undetectable manner [13], and operating system channels using file lock states [1]. A complete list of such studies using various resources is shown in Table 1. A recent study designed a real world malware app that uses audio as a covert channel [29]. The authors mention the use of various system settings as possible covert channels including volume, vibrator and screen. Another related study again analyze the use of system settings [14] as covert channels including volume, vibrator and file locks to provide a threshold based detection mechanism. A survey of various covert channels in [19] demonstrate possible timing and storage channels. This study enumerate most of the storage and timing channels. However, they do not perform a systematic study on all shared resources and its properties. In addition to covert channels, overt channels have also been studied in [22], where a legitimate communication medium is misused to encode malicious information.

In order to perform a systematized analysis of shared resource properties that can support covert channels, we enumerate all shared resources in an Android device, and identify attributes that satisfy covert channel properties [3]. These properties are as follows:

1. Both the sending and receiving processes must have access to the same attribute of a shared object.
2. 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.
3. The receiving process must be able to reference that attribute of the shared object.
4. The sending process must be able to control the detection time by the receiving process, for a change in attribute value.

5. A mechanism for initiating both processes, and properly sequencing their respective accesses to the shared resources, must exist.

Table 1 shows a list of shared resources typically available to each app as part of the current Android system. Shared resources are classified as application, system and hardware level resources [19] corresponding to its attribute properties as seen by an Android app. Previous studies have already identified many shared resources that could form a covert channel. This is summarized in the table, represented by ✔ besides corresponding shared resources (along with references). However, it can be seen that the hardware resources such as Battery and Phone have not been analyzed to form a covert channel (indicated by ✗).

In addition to identification of shared resources, there have been multiple approaches proposed for detecting the covert and overt channels. Authors in [9] implement an information flow tracking system for data and IPC, called TaintDroid. The system uses taint tags for detecting data flow between source and sink. A more relevant detection system is XManDroid [4], which uses a mapping structure to track interactions between two apps using shared resources, by user policy enforcement. This system overcomes the drawbacks of TaintDroid in tracking shared resource such as system settings. However, it is prone to high false positives and requires a large set of user defined policies for each app installed on the device.

In this report, we focus on systematically analyzing shared hardware resources on an Android device to demonstrate a possible high throughput covert channel using their attributes. During our analysis, we discover two new covert channels using Battery and Phone component. Further, we also enrich the existing work that use Screen and Volume as a covert channel by using certain attributes not discussed in these earlier studies. We design and evaluate a communication protocol, to form a high throughput channel by using synchronization methods (with empirically determined thresholds in case of timing channels), and demonstrate the channel feasibilities.

3 Analysis Overview

In this section, we show a methodology to identify shared resources (using a shared resource matrix [15]), and inspect each of these resources to form a covert channel between colluding apps, within a device.

Hardware resources are inherently shared by various apps in a device. A systematic approach that identify shared resources which support covert communication between colluding malicious apps performs analysis on its attributes that satisfy all properties of a covert channel. In general, the following properties of shared resource attributes are desirable.

- Ability for apps to *read* from a resource.
- Ability for apps to *write* to the same resource.
- Ability for apps to turn *on* (or *off*) a resource.

		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?		○	○	○	●	◐	○	◐	●	○	○	○	-

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.)

- Ability for an app to *lock* a resource, in which case other apps cannot access the same resource simultaneously.

In this study, we enumerate each hardware shared resource on an Android device and create a matrix to identify attributes that satisfy these properties. This matrix is presented in Table 2, which lists hardware resources as per Table 1. We analyze and construct a covert channel between two apps using resources that have attributes that satisfy both *read* and *write* properties, *lock* property, or *on/off* property. These shared resource attribute properties provide all required covert channel properties to the colluding apps as listed in Section 2. A communication protocol using these shared resources are implemented to demonstrate a data transfer between two colluding apps.

From the table, we found 11 out of 12 hardware resources listed that have attributes with desirable properties. This includes resources such as Screen, Volume, Phone, CPU, Memory and Battery having attributes to satisfy *read* and *write* properties. In addition, resources such as Camera, Phone and Bluetooth have attributes satisfying *lock* property, and those such as Network (WiFi and Cellular data), Bluetooth, Sensors, Vibrator and GPS attributes that satisfy *on/off* property to form a single bit covert channel. Here, the Phone resource denote the system used to make phone calls. Among these, resources such as CPU, Memory, Sensors, Vibrator, Camera, Bluetooth and Network have already been identified in earlier studies (denoted by ○ in Table 2). Hence, we do not consider them for our analysis. Further, USB does not satisfy any of the desired properties on its own. An external action such as USB attachment may be required. Hence we do not analyze this resource. Interestingly, we do find two new covert channels (denoted by ●), namely Battery and Phone component (details in Section 4). Also, for the two previously studied covert channels (denoted by ◐), we enrich them by using attributes not specified in those studies (details in Section 5).

4 Discovering New Covert Channels

With our systemized analysis, we have identified two new covert channels: Battery and Phone call. In this section, we present our discovery.

4.1 Battery

The energy needed for all resources to perform their tasks is provided by the battery in a device. Each resource require a certain amount of power for their operation which discharges the battery at a corresponding rate. Parallel use of resources on a device can form a covert channel where the rate of battery discharge indicates an encoding of desired binary data. The decoding app measures this battery discharge rate to decode the information.

Mobile devices typically have a Lithium-ion battery, with limited charge capacity. Parallel use of multiple resources discharge the battery at different rates, depending on the resources used. Applications can register a receiver (intent filter registered with `ACTION_BATTERY_CHANGED`) to the `BatteryManager` API which provides a broadcast intent [24] informing the app about every 1% change in the battery charge level. A malicious app can perform a binary encoding by running parallel operations on combination of resources such as CPU, screen brightness, cellular data, GPS and audio to achieve a predetermined battery discharge rate representing a binary digit. A decoder participating in the collusion receives the broadcast intent at the same rate. This can be used to estimate the battery discharge rate for an exact time period, and decode the desired binary digit based on the same predetermined threshold. This is a timing channel since the protocol require exact time synchronization between the two apps.

4.2 Phone Call

The `TelephonyManager` API can be used to access the phone component attributes by an app. This provides the app an ability to make phone calls, which can be used to form a covert channel.

With appropriate permissions (`CALL_PHONE`), apps can make a phone call using an intent with `ACTION_CALL`, and end the call using a Java reflection method involving the `ITelephony` interface. In addition, the state of the phone component can be read by apps having `READ_PHONE_STATE` permission. The state of the phone such as `CALL_STATE_OFFHOOK`, `CALL_STATE_IDLE`, etc is provided by the API informing of a change in call state [28] to an app that registers a receiver to the broadcast intent. There are two ways in which such an ability to make a phone call can be used as a covert channel.

Phone Call Frequency Channel : A malicious app (decoder) can register to receive the broadcast intent informing call state changes. The encoder app then can place calls (and end it quickly) at a certain predetermined frequency that can encode a binary digit. As a consequence, the decoder app receives broadcast intents informing a change of phone state, at the same rate. By measuring the

rate at which `CALL_STATE_OFFHOOK` changes for the exact time period, the binary digit can be decoded based on the same predetermined threshold. Since both colluding apps require synchronization to obtain a correct call state for the exact time period, this is a timing channel.

Phone Call Log Channel : Here, we use the phone call log as a storage attribute to transfer data covertly. Apps can place an integer string encoding of desired information in the `URI` attribute of a phone call intent. As before, a call is placed using this intent, and is ended instantly. The dialed number is stored in the 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 [25]. 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 Enriching Existing Covert Channels

We reported in Table 2 that there are also existing efforts (e.g., [14, 19, 29]) 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.

5.1 Screen

Screen system setting parameter attributes are accessible by all apps. This can be used to form a covert channel where an app sets the screen parameter for encoding required information, as shown in [29] and [19]. Here, we analyze a specific attribute namely `SCREEN_BRIGHTNESS` system settings parameter, which is not mentioned in early studies.

In order to form a covert channel, an encoder can set the `SCREEN_BRIGHTNESS` parameter with an appropriate value to encode the required information. The decoder can then read this value to obtain the information. Specifically, screen brightness can be changed to an appropriate integer value in the range of 0 to 255 by an app [27] when the `SCREEN_BRIGHTNESS_MODE` parameter is set to 0. Integers in this range corresponds to a binary string of length 8 bits ($2^8 = 256$). Therefore, an encoder can convert an information of length 8 bits to an integer which can be set as a screen brightness value.

5.2 Audio (Volume)

Prior efforts (e.g., [19,29]) identified the audio channel using system settings and APIs involving the volume attribute. A covert channel is formed by setting an appropriate value to the volume settings in a similar manner as screen brightness

Volume Components	Abbr.	Range	Bit Length
STREAM_MUSIC	M	0 - 15	4
STREAM_DTMF	D	0 - 15	4
STREAM_ALARM	A	0 - 7	3
STREAM_RING	R	0 - 7	3
STREAM_NOTIFICATION	N	0 - 7	3
STREAM_SYSTEM	S	0 - 7	3
STREAM_VOICE_CALL	V	0 - 5	3

Table 3. Volume Stream Setting AudioManager API

shown earlier. Here, we provide new insights regarding the use of multiple audio 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` [23]. 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. This is shown in Table 3 with the corresponding binary bit length of information that can be encoded into the component. A bit length is calculated as $log_2(\#range)$, where $\#range$ is the number of discrete values in the range. For example, there are 16 discrete values in the range for component M. Therefore, its bit length is $log_2(16) = 4$ i.e. 4 bits can be used to encode any of the 16 discrete values. A binary encoding scheme can combine these components to form a single covert channel by concatenating all components, each encoding a binary string of length equal to its corresponding bit length. The total length of this binary string is 23 bits, which is the sum of bit length of all components listed in Table 3. Decoding is performed by reading all component values and concatenating it in the same manner. Interestingly, we observed a limitation on using all components to form a covert channel, whose details are provided later in Section 7.

6 Protocol Design

In this section, we design a protocol that we use for covert communication to maximize throughput and minimize error during data transfer between two malicious colluding apps on the same device. A typical communication protocol involves the use of control channels which may either form a part of the overall channel bandwidth reducing the data throughput, or may be use a separate channel. In our design, we choose to use a different covert channel to for each control channel while only the encoded data is transmitted over a data channel. This design choice provides a higher throughput and better coordination of en-

coding and decoding process than using it as part of the channel used for data encoding (data channel), due to the following limitations.

- Limited channel storage capacity in a storage channel.
- Inability to append information to the existing value.
- Changes made by the user or other apps installed on the device

The control information between encoder and decoder apps are provided to overcome challenges due to channel errors. These challenges are as follows.

- Noise from external entities such as parallel execution of apps on the same device, or user interaction.
- Scheduling uncertainty of intents and call back functions that may be required in a communication protocol.
- Bandwidth Limitation of the shared resource attribute used to form the covert channel.
- Non synchronization of encoding and decoding process.

In order to overcome these challenges, we use a data splitting, synchronization and a protocol disruption mechanism for data transfer between the two colluding apps. In particular, a typical data size for private data such as contact information is around 2.5 kilobytes (as seen on our test phone). As seen in previous sections, shared resources have limited bandwidth. For example, screen brightness can be set to an integer value corresponding to a binary string of length 8 bits. Therefore, the data is split into multiple chunks of appropriate size (e.g. 8 bit chunk size if screen brightness channel is used). Each chunk is transmitted sequentially from the encoder to the decoder addressing the channel bandwidth limitation. The sequential data transmission requires some synchronization between the encoder and decoder apps. If a storage channel is used, the encoder should write the data to a channel attribute before it is read by the decoder. Also, the encoder should write a next data chunk after the decoder completes reading the previous data chunk. In case of a timing channel, the encoder and decoder should have a precise time synchronization during the encoding process. Therefore, noise due to non-synchronization is addressed using a synchronization protocol. Further, the use of multiple covert channels for data and control signals can be affected due to external entities that may change the attribute value used for a channel (control or data) disrupting the protocol or inducing decoding errors. For such cases, the communication protocol monitors each channel by checking the previously written channel information (in case of storage channels used for data and control signals) before encoding a new information. We use another clocking mechanism called *session clock*, similar to the synchronization protocol, to indicate start and end of data transfer for apps to end an encoding if external changes are detected. Finally, timing channels require thresholds to encode the binary values of 0 and 1. We empirically determine these thresholds for each timing channel providing an error margin to address scheduling uncertainty and noise due to external entities. Further, error detection schemes [21] can also be used for better error handling. However,

they can only be used for multi-bit channels and reduce effective throughput. We leave this for future work.

6.1 Communication Protocol

We define an *event* as a single encoding of a data chunk using the channel. Assuming large data size, transmission of complete data from an encoder to a decoder app require multiple *events*. In order to maximize the throughput (number of bits transmitted per second) of the channel, we use a clocking mechanism, called *event clock*, which aids in ordering of events. Similarly, we define a *session* as a set of *events* that occur sequentially to transmit the complete data. The clocking mechanism for a session is called *session clock*, which indicates the start and end of data transmission between the apps.

In order to begin synchronization between the two colluding apps, an initialization protocol is required. Initialization protocol also uses a covert channel to avoid detection. In this report, we use a single covert channel where the encoder app (or sender) performs a handshake protocol (such as sending a shared private key) to establish communication with a decoder app. The decoder app (or receiver) responds with an answer using the same channel. The encoder app begins data transmission upon realizing a correct response from the decoder.

Algorithm 1 shows a generic protocol for covert communication. The encoder is assumed to transmit a binary data of total length N. Let k be the number of bits transmitted per event. Encoder splits the N length binary data to k bit binary data chunks. The *event clock* is initially set. Encoder sets a *session clock* to indicate beginning of a session. In the case of a storage channel, the encoder encoders a data chunk (*split*) and then unset's (value of 0) the *event clock* to indicate *event* completion. The encoder then waits for the decoder to set the *event clock*. The decoder waiting for the *event clock* to unset, then decodes a value from the channel. It checks if the *event clock* is set and *session clock* is unset. If this is true, the value is discarded and the decoding session is ended. If this is evaluated to false, the decoder set's the *event clock* to indicate finish of data read. The decoder waits for the next event or the end of session. The encoder then checks for any disruption of data channel by reading the previously written value. In case of a change in the value, the *event clock* is set and the *session clock* is unset indicating protocol disruption. Both encoder and decoder then ends its operation, to restart at a later stage. If the previous value of the channel written by the encoder has not changed, then encoding process continues. The *session clock* is unset by the encoder after transmitting all N bits. Finally, the original value of the parameters (*event clock* and *session clock*) are replaced after the decoder decodes the last binary data segment. This synchronization aids in minimizing error in communication.

This protocol can be used for both storage and timing channel. However, in case of a timing channel, the decoder begins to decode once the *event clock* is set, indicating the start of encoding. The *event clock* is only controlled by the encoder to signal the change in event to the decoder. The decoder does not change the value of the *event clock*. This change in behavior of *event clock*

Algorithm 1: Covert Communication Protocol for Data Transfer

Require: Initialization Protocol complete.
Ensure: Input in binary form of length N

Encoder Operation:

origEveValue ⇐ *event clock*
origSesValue ⇐ *session clock*
Save original channel parameter value(s).
session clock ⇐ 1
event clock ⇐ 1
Initialize channel parameters if necessary
split ⇐ Split binary input into k bit ensemble converted to integer form.
for *split* **do**
  if (Storage Channel) **then**
    Encode *split* in channel parameter(s)
    event clock ⇐ 0
    Wait till *event clock* = 1.
  else
    event clock ⇐ 0
    Encode *split* in channel parameter(s)
    event clock ⇐ 1
  end if
  Check for change from previous channel value
  if (Change detected) **then**
    session clock ⇐ 0
    event clock ⇐ 1
    Break.
  end if
end for;
if (Storage Channel) **then**
  Wait till *event clock* = 1.
else
  event clock ⇐ 1
end if
Sleep for 5 secs.
session clock ⇐ 0
event clock ⇐ 0
event clock ⇐ *origEveValue*
session clock ⇐ *origSesValue*
Reset channel parameters to initial value(s)

Decoder Operation:

Wait for *session clock* = 1.
while *session clock* = 1 **do**
  if (*event clock* = 0) **then**
    if (Storage Channel) **then**
      Read channel parameter value.
      event clock ⇐ 1
      Decode value.
    else
      Measure parameter till *event clock* = 1
      Decode value using threshold.
    end if
    if (*event clock* = 1 & *session clock* = 0) **then**
      Discard value
      Break.
    end if
    Save value.
  end if
end while

control is to accommodate the exact time synchronization required in a timing channel.

7 Implementation

This section provides a detailed description of our implementation of initialization and data transfer protocols, using shared resources. Implementation require

the use of multiple shared resources for the two synchronization clocks (*session* and *event* clock), and a data transfer channel. The *session* and *event* clock can be implemented using a single bit covert channel, where 'set' corresponds to 1 and 'unset' corresponds to 0. Single bit covert channels include system setting parameters such as `ACCELEROMETER_ROTATION`, `HAPTIC_FEEDBACK_ENABLED` and `VIBRATE_ON`. We use two of these parameters to emulate the clock behavior. Additionally, the protocols are implemented to run as a background service in order to make it oblivious to the user as much as possible.

7.1 Initialization Protocol

The initialization protocol involve a shared secret key exchange between the colluding applications, using a covert channel such as `SCREEN_BRIGHTNESS` system settings parameter. Without loss of generality, we use a static binary string as a secret key. The encoder changes screen brightness value rapidly, encoding a secret key in successive *events*. If a decoder detects these changes, it responds by repeating the decoded message using the same channel. The decoder then starts listening to a pre-defined covert channel for data from the encoder. The encoder waiting for an acknowledgement, observes the echo to conclude the presence of a decoder application and starts data transmission.

7.2 Data Transfer Protocol

Storage Channel : Storage channels uses the implicit storage property of the attribute for data communication. When using the phone call log channel, encoding is performed by placing an integer string (data segment considered for an *event*) in the `URI` attribute of the `TelephonyManager` class. The encoder sends an intent to place a phone call, and then ends the call immediately by accessing the private `endCall()` method using `ITelephony` object. Decoder reads the call log after the encoder unsets the *event clock*, to obtain the latest called number. In this case, the complete binary data is converted to an integer string by the encoder. Therefore the *session clock* is unset after the decoding process is complete.

In case of the volume (or audio) channel, the `AudioManager` object is used to access volume properties for encoding binary information. We observe multiple limitations of using all the volume components to form a single covert channel. We observed that the `STREAM_SYSTEM`, `STREAM_RING` and `STREAM_DTMF` components are dependent on each other. Any change made to `STREAM_DTMF` value affects the`STREAM_SYSTEM` and `STREAM_RING` values. Figure 2 shows this resultant volume level when varying the `STREAM_DTMF` volume level. In addition, we observed that a system setting can link `STREAM_RING` and `STREAM_NOTIFICATION` volume on our test phone.

In order to overcome this limitation, we use `STREAM_DTMF` for its higher capacity (4 bits of information per event), and ignore `STREAM_RING` and `STREAM_SYSTEM` volume. For simplicity, we also ignore `STREAM_VOICE_CALL` volume as it does not

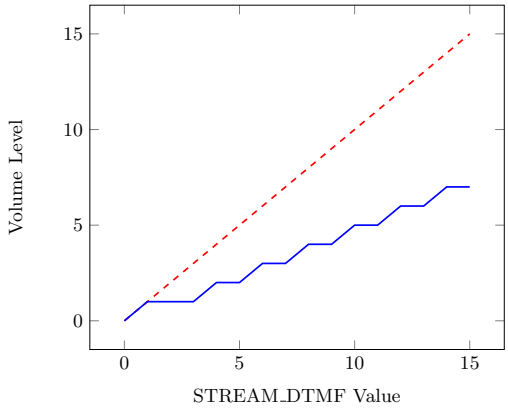

Fig. 2. Volume Component: - - - STREAM_DTMF; —— STREAM_SYSTEM and STREAM_RING

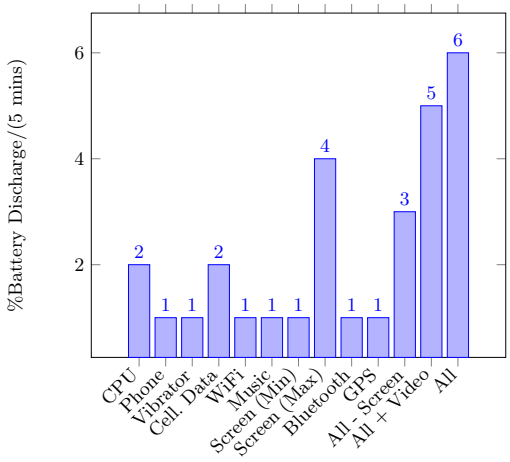

Fig. 3. Component Battery Discharge Level

represent a complete 3 bit information. This reduces the total capacity of the channel to 14 bits per event as shown in Table 5.

Timing Channel : Timing channel typically use thresholds for encoding information onto the channel. In case of a battery channel, discharge rate used as a threshold to encode binary information i.e. encode 1 if drain rate crosses a certain threshold, or a 0 it is within certain threshold limits, depends on a phone's hardware power limitations and overall battery health [11]. A systematic determination of achievable discharge rates on our test phone (Samsung Galaxy S) was performed by evaluating the maximum discharge rate of each component. Figure 3 shows maximum battery discharge percent (averaged over 15 mins) for each component over a period of 5 mins, on the test phone. Here, *ALL* indicates

Components	Description
CPU	10 threads running parallel continuously performing a random 1000×1000 matrix multiplication.
Cellular Data	Download 10MB file in a loop.
WIFI	Download 10MB file in a loop.
Music	Play a MP3 file using the music player continuously.
Screen	Full or Dim Brightness level.
Vibration	Enable vibrator to vibrate at a rate of 1 sec.
GPS	Continuously poll the longitude and latitude coordinates.
Bluetooth	Continuously poll for devices to connect.
Phone	Continuously call a specific telephone number.
Video	Continuously play a video file.

Table 4. Components considered for Battery evaluation

use of components including CPU, Cellular Data download, Music, Max Screen Brightness, Vibration, GPS, Bluetooth and Phone Call as per Table 4, in parallel. A lower threshold L_h is used to encode 1, and a $[(U_l), (L_l)]$ threshold tuple indicating upper and lower level threshold to encode 0, where $L_h > U_l > L_l$. We use max screen brightness, download (using 4G), GPS, Bluetooth and Music to achieve a discharge rate of at least $L_h = 0.01$ to encode 1. Similarly we use min screen brightness, CPU and Music to achieve a discharge rate with at most $U_l = 0.008$ and $L_l = 0.004$ to encode 0. These discharge rate threshold values are determined empirically.

In a similar manner, encoding in the phone call frequency channel is performed by placing phone calls at a certain rate. We add a delay in placing subsequent calls to achieve different call frequencies. The threshold $L_h = 0.5$ was used for encoding 1, and that for encoding 0 were $U_l = 0.5$ and $L_l = 0.12$. These thresholds were also empirically determined to achieve negligible decoding error under normal operating conditions.

8 Evaluation

In this section, we provide the experimental results of the covert channels analyzed. The evaluation of each channel was performed on a Samsung Galaxy S phone running Android version 4.2.2 (test phone). The encoder and decoder apps use minimal permissions including `INTERNET`, `BLUETOOTH`, `ACCESS_FINE_LOCATION`, `WAKE_LOCK`, `CALL_PHONE`, `READ_PHONE_STATE` and `READ_CALL_LOG`, depending on the covert channel used. Experiments involve data transfer of a random binary string of certain length (given under *Input Length* column in Table 5), from an encoder to a decoder using each covert channel mentioned in Section 4 and Section 5. As mentioned in Section 7, the binary string is divided into data chucks of appropriate size (given under *Binary Length* column in Table 5) for each channel.

Covert Channel		Supported Range		Input Length L (bits)	Time Taken T (sec)	Throughput L/T (bps)
		Integer Range	Binary Length			
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

Table 5. Protocol statistics with *Throughput*: Ratio of Input Length (L) in bits and Time Taken (T) in seconds

Overall Result Table 5 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 channe as shown in Figure 4. This shows that 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 [19] and [14] respectively. This is primary due to the use of higher bandwidth attribute(s) to form the channels. In case of volume channel, we obtain an average throughput of 328 bps using synchronization, as compared to a throughput of 101 bps obtained in [14] which does not use synchronization. This is a higher bandwidth channel that we construct compared to previous approaches. As mentioned in Section 5, we use multiple components, each supporting an integer value whose corresponding binary string is of length 3 or 4. Similarly, in the case of screen channel, existing effort [19] reported an average of 65.89 bps for single settings which use a single setting channel used to switch on/off the screen. Our implementation yield a throughput of 634 bps where encoding of 8 bits can be performed at once (converting the integer range supported to corresponding binary digits). This single setting used in our approach provides a better throughput than previously reported. Additionally, Table 5 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

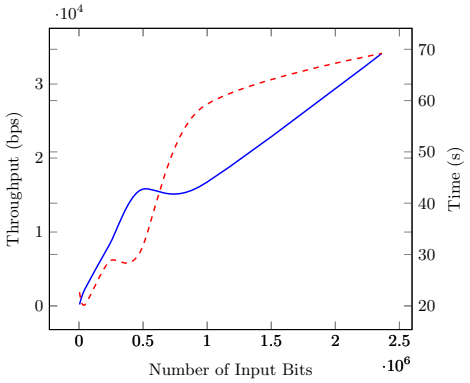

Fig. 4. Phone Call Log: —— Throughput (bps); - - - Average Time (s)

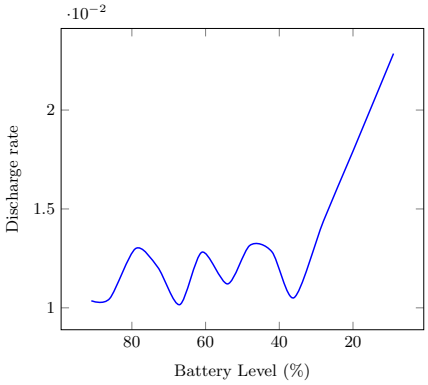

Fig. 5. Battery Discharge Behavior: —— Discharge Rate

the Screen channel. We obtained a throughput of 105 bps when using only a single Volume component (i.e. `STREAM_DTMF` to transfer 525 bits of information indicating a slow response time.

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 Section 4. Figure 6 shows an example binary input in the phone call frequency channel, with the corresponding frequency decoded at the decoder. The threshold of 0.5 is shown in the figure, indicating the measure (L_h) used for decoding the binary value. During the experiments, we found that these intents are not scheduled at a desired frequency by the intent handler, as shown in the example figure. For instance, the input at count 20 is 1. Its corresponding frequency evaluated by the decoder is 0.946 calls per second. This is clearly not the highest frequency obtained for encoding 1. This is due to interference from multiple process calls generated for handling each intent generated for each phone state change. We found that this latency decreased significantly with delays in phone calls being placed by the encoder.

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. Figure 5 shows the battery discharge rate achieved on a full battery discharge cycle, with an interval of 5 mins per *event*. The figure shows that the battery discharges at a faster rate when its level reaches about 30% of full capacity. We observed this behavior in multiple runs of discharge cycle, which is consistent with a Lithium-ion battery property [11]. Therefore,

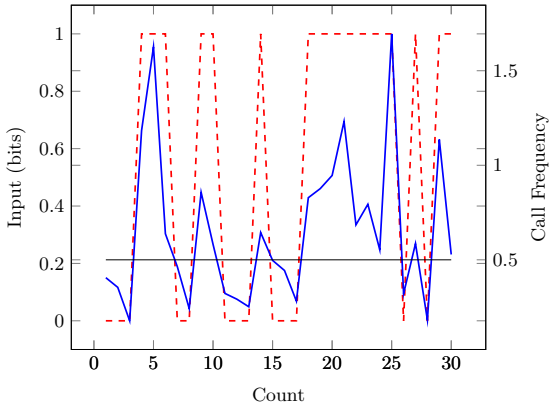

Fig. 6. Phone Call Frequency Example: - - - Input Data; —— Decoded Frequency

		Data Transmission Channel				
		Volume	Vibrator	Accelerometer	Haptic Feedback	Screen Brightness Mode
Event Clock	Vibrator	329	X	10	46	37
	Accelerometer	253	4	X	4	3
	Haptic Feedback	325	37	9	X	31
	Screen Brightness Mode	270	32	11	31	X

Table 6. Throughput (bits per sec) with different *event clocks*

in our evaluation of an attack scenario involving the battery, we do not use the channel if the battery charge level is below 30%.

Synchronization Clock We further evaluate the use of single bit channels for *session clock* and *event clock*. Table 6 shows various throughput that was achieved using our test phone, on using various single bit parameter over different data transmission channels. Each throughput was calculated by averaging over 3 transmission sessions with 525 bits per session. Interestingly, we observed that use of different system settings attributes as *event clock* provided different throughput. The table shows that volume channel provides the highest throughput overall when using `VIBRATE_ON` as the *event clock*, and a lower throughput when using `ACCELEROMETER_ROTATION` system setting, for all channels. This behavior may be attributed to process scheduling and hardware limitation on the device. In the table, an X represents an invalid scenario since the same channel cannot be used for both clock and data transmission.

9 Discussion and Future Work

In this section, we discuss various challenges and limitations of using these covert channels. Error during decoding of information from a covert channel may be due to internal channel errors caused by non-synchronization, external factors such as channel usage by other unsuspecting apps or user interactions. When designing a protocol that does not use any synchronization bit, these errors are challenging to address. In this case, one way to overcome errors is to learn the channel usage over time. However, when using a synchronization bit, the encoder and decoder can be made aware of external changes to the resource parameters involved in the protocol, and check for parameter modification external to the protocol sequence. This significantly reduces errors since either the encoder or decoder app can raise an alarm to abandon the transmission session. One limitation of our implementation on this regard is that this alarm is raised for any change in the data channel only. If a change is made to the control channel intelligently, our protocol may be broken. However, we leave a better adaptation of such implementation for future work. In the case of large bandwidth channels, the error detection and correction system can be included as part of data encoding, for better error handling. Our experiments show that synchronization between colluding apps are required to overcome channel error. We evaluated the covert channels without the use of any synchronization bits. In this case, the encoder and decoder exchange data at a certain frequency. We observed an erratic behavior in terms of throughput achieved by the system. This is as expected, where certain experiments resulting in a higher transmission rate, induces more error while decoding the data.

Previous studies have shown various methods to detect the use of covert channels [4]. However, the major limitation of these systems are high false positives, specific policies that need to be customized for each user or application, or changes in the Android middleware for its functioning. Further, the malware designer may use channel obfuscation to evade a detection system. For example, battery channel can be used to choose a channel for data transmission, which uses a certain discharge rate decided by a randomized algorithm (involving a protocol similar to Diffe-Hellman key exchange [7]). Detection of timing channels by these systems are currently not possible.

We show an existence of high bandwidth Phone Call Log channel. This is due to string length check not performed on dialed number. The bandwidth of this channel can be reduced by limiting the string length of each log record. One limitation of covert channel involving the phone component is that the channel usage cannot be made oblivious to the user. The user can detect a phone call being made, or review the phone call log for random dialed numbers. We leave the evaluation of channel obfuscation to avoid detection, for future work.

10 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.

References

1. Ali, Mohammad, Humayun Ali, and Zahid Anwar. "Enhancing Stealthiness & Efficiency of Android Trojans and Defense Possibilities (EnSEAD)-Android's Malware Attack, Stealthiness and Defense: An Improvement." Frontiers of Information Technology (FIT), 2011. IEEE, 2011.
2. Barrera, David, H. Gne Kayacik, Paul C. van Oorschot, and Anil Somayaji. "A methodology for empirical analysis of permission-based security models and its application to android." In Proceedings of the 17th ACM conference on Computer and communications security, pp. 73-84. ACM, 2010.
3. Bishop, Matt. Introduction to computer security. Addison-Wesley Professional, 2004.
4. Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. "Xmandroid: A new android evolution to mitigate privilege escalation attacks." Technische Universitt Darmstadt, Technical Report TR-2011-04 (2011).
5. Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards Taming Privilege-Escalation Attacks on Android." In NDSS. 2012.
6. Chin, Erika, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing inter-application communication in Android." In Proceedings of the 9th international conference on Mobile systems, applications, and services, pp. 239-252. ACM, 2011.
7. Diffie, Whitfield, and Martin E. Hellman. "New directions in cryptography." Information Theory, IEEE Transactions on 22.6 (1976): 644-654.
8. Ehringer, David. "The dalvik virtual machine architecture." Techn. report (March 2010) (2010).
9. Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones." Communications of the ACM 57, no. 3 (2014): 99-106.
10. Enck, William, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. "A Study of Android Application Security." In USENIX security symposium, vol. 2, p. 2. 2011.

11. Ferreira, Denzil, Anind K. Dey, and Vassilis Kostakos. "Understanding human-smartphone concerns: a study of battery life." Pervasive Computing. Springer Berlin Heidelberg, 2011. 19-33.
12. Fuchs, Adam P., Avik Chaudhuri, and Jeffrey S. Foster. "SCanDroid: Automated security certification of Android apps." Manuscript, Univ. of Maryland (2009).
13. Gasior, Wade, and Li Yang. "Network covert channels on the Android platform." Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research. ACM, 2011.
14. Hansen, Michael, Raquel Hill, and Seth Wimberly. "Detecting covert communication on Android." Local Computer Networks (LCN), 2012 IEEE 37th Conference on. IEEE, 2012.
15. Kemmerer, Richard A. "Shared resource matrix methodology: An approach to identifying storage and timing channels." ACM Transactions on Computer Systems (TOCS) 1.3 (1983): 256-277.
16. Lalande, Jean-Francois, and Steffen Wendzel. "Hiding Privacy Leaks in Android Applications Using Low-Attention Raising Covert Channels." In Availability, Reliability and Security (ARES), 2013 Eighth International Conference on, pp. 701-710. IEEE, 2013.
17. Lampson, Butler W. "A note on the confinement problem." Communications of the ACM 16.10 (1973): 613-615.
18. Marforio, Claudio, and Aurlien Francillon. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Department of Computer Science, ETH Zurich, 2011.
19. Marforio, Claudio, Hubert Ritzdorf, Aurlien Francillon, and Srdjan Capkun. "Analysis of the communication between colluding applications on modern smartphones." In Proceedings of the 28th Annual Computer Security Applications Conference, pp. 51-60. ACM, 2012.
20. NCSC, NSA. "Covert Channel Analysis of Trusted Systems (Light Pink Book)." NSA/NCSC-Rainbow Series publications, (1993).
21. Michelson, Arnold M., and Allen H. Levesque. "Error-control techniques for digital communication." New York, Wiley-Interscience, 1985, 483 p. 1 (1985).
22. Ritzdorf, Hubert. Analyzing Covert Channels on Mobile Devices. Diss. Master Thesis ETH Zrich, 2012, 2012.
23. `http://developer.android.com/reference/android/media/AudioManager.html`
24. `http://developer.android.com/reference/android/os/BatteryManager.html`
25. `http://developer.android.com/reference/android/provider/CallLog.Calls.html`
26. `http://source.android.com/devices/tech/security/`
27. `http://developer.android.com/reference/android/provider/Settings.System.html`
28. `http://developer.android.com/reference/android/telephony/TelephonyManager.html`
29. Schlegel, Roman, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones." In NDSS, vol. 11, pp. 17-33. 2011.
30. Simon, Laurent, and Ross Anderson. "PIN skimmer: inferring PINs through the camera and microphone." Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices. ACM, 2013.
31. van Cuijk, WP Mark. "Enforcing a fine-grained network policy in Android." (2011).

32. Vetter, J., P. Novak, M. R. Wagner, C. Veit, K-C. Mller, J. O. Besenhard, M. Winter, M. Wohlfahrt-Mehrens, C. Vogler, and A. Hammouche. "Ageing mechanisms in lithium-ion batteries." Journal of power sources 147, no. 1 (2005): 269-281.
33. Zhang, Lide, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. "Accurate online power estimation and automatic battery behavior based power model generation for smartphones." In Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pp. 105-114. ACM, 2010.