Colluding Covert Channel for Malicious Information Exfiltration in Android Environment

Rosangela Casolare[1], Fabio Martinelli[3], Francesco Mercaldo[2,3] and Antonella Santone[2]
[1]*Department of Biosciences and Territory, University of Molise, Pesche (IS), Italy*
[2]*Department of Medicine and Health Sciences "Vincenzo Tiberio", University of Molise, Campobasso, Italy*
[3]*Institute for Informatics and Telematics, National Research Council of Italy, Pisa, Italy*

Keywords: Android, Security, Model Checking, Formal Methods, Privacy.

Abstract: Mobile devices store a lot of sensitive and private information. It is easy from the developer point of view to release the access to sensitive and critical assets in mobile application development, such as Android. For this reason it can happen that the developer inadvertently causes sensitive data leak, putting users' privacy at risk. Recently, a type of attack that creates a capability to transfer sensitive data between two (or more) applications is emerging i.e., the so-called colluding covert channel. To demonstrate this possibility, in this work we design and develop a set of applications exploiting covert channels for malicious purposes, which uses the smartphone accelerometer to perform a collusion between two Android applications. The vibration engine sends information from the source application to the sink application, translating it into a vibration pattern. The applications have been checked by more than sixty antimalware which did not classify them as malware, except for two antimalware which returned a false positive.

1 INTRODUCTION

The large-scale spread of Android based devices has led to the implementation of a large number of malicious software (i.e., malware) aimed at stealing confidential information. Sensitive data, managed and stored inside our smartphones, attract the attention of malicious developers that, by exploiting the weaknesses of the security features provided by the operating system developed by Google and taking advantage of users' carelessness, can cause great damage.

In May 2020, there were about 430,000 malware attacks on Android devices, a 3.6% increase over the previous month. Instead, in August 2020 the growth was 6.26% compared to July[1].

Android is the operating system in mobile environment most popular, with a market share about 74.6% [2]. This feature combined with being open source, makes Android interesting in the eyes of cybercriminals, because by rebuilding the source code it is possible to create customized operating systems (Enck et al., 2014; Enck, 2011). There is also the possibility of installing applications from third-party stores, but downloading applications from sources of unknown origin is very dangerous, as they are not subject to the security controls present in the official stores (for Android it is Google Play Store), even if malicious applications may also be present in the official stores, in smaller quantities than to unofficial ones (Nguyen et al., 2020; Canfora et al., 2018).

Recently, cybercriminals are working for developing new threats to perpetrate more and more harmful actions, with the aim to evade the current free and commercial antimalware, mainly signature-based (Cimitile et al., 2018; Mercaldo et al., 2016a). One of these new threats is represented by the *collusion attack*. The rationale behind this emerging attack is to split the malicious action in two or more applications able to perform a communication for sensitive data exfiltration and sharing (Casolare et al., 2020a). This type of attack avoids that an application with too many permissions being immediately reported by the security mechanisms (i.e., antimalware) or that makes the user suspicious (Mahboubi et al., 2017; Cimino et al., 2020; Mercaldo et al., 2016b).

[1]https://news.drweb.com/show/review/?i=13991&lng=en

[2]https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

Casolare, R., Martinelli, F., Mercaldo, F. and Santone, A.
Colluding Covert Channel for Malicious Information Exfiltration in Android Environment.
DOI: 10.5220/0010396708110818
In *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*, pages 811-818
ISBN: 978-989-758-491-6

As matter of fact, to access to user sensitive resources stored in the devices (for instance the contact list or the device localisation), the developer must explicitly declare the related permissions in the application. To avoid a single application having all accesses guaranteed to complete the attack and therefore being classified as a threat from the antimalware, it is possible to split the malicious action (i.e., the permission request) into multiple applications which will then ”complete” each other. For example, an application could have the authorization to read sensitive data but not the one to access the internet, so it would not represent any threat to the user and would come out unscathed from an antimalware scan. Similarly, an application with network access permissions but without the ability to access sensitive data would not arouse suspicion.

If the two applications are performing a collusion, they will be able to collect the data and send it over the network without the attack being intercepted. To ensure that transmission between applications occurs undetected, it is important to create a hidden, undetectable communication channel.

This type of attack is called *covert channel* precisely because it allows data to be transferred through a channel not designed to transmit information, but which can be used for this purpose to hide communication (Shrestha et al., 2015). The advantage is that, unlike communication channels, covert channels are not subjected to the control and security mechanisms of the operating system, making transmission particularly difficult to identify.

In this article we design and we implement the colluding cover channel attack in Android environment. To do this, we exploit the vibration sensor, present on all mobile devices, as a communication channel to send sensitive information between applications installed on the same device. To test the functioning of the communication, three different real-world smartphones have been considered, in order to validate the correctness of the work.

The goal of this work is to demonstrate the effectiveness of this attack and to make available to the research community a data-set that performs this type of colluding attack, which can be downloaded for research purposes at the following link: https://mega.nz/folder/E0wwRABD#YM7U7sru5ZvD6ijsCbKH4Q.

The paper’s organization is the following: in section 2 we describe the different types of covert channels and their functionality; in section 3 we explain how is composed the proposed data-set and in particular which is the process to obtain the necessary permissions; section 4 shows how data are converted and sent, and the results obtained with antimalware analysis; in section 5 current state-of-the-art literature is analyzed and, finally, conclusion and future research lines are drawn in section 6.

2 UNDERSTANDING THE COVERT CHANNEL COMMUNICATION

Android implements a permissions-based security model in order to limit the actions that each application can perform. A collusion attack is able to ”break” this model, distributing the permissions it needs across multiple applications.

Applications are classified as *source* when they allow information to escape from the device thanks to read permissions, instead they are classified as *sink* when they receive data and send it outside thanks to connection permissions.

The covert channels are those channels not intended for the transfer of information, which can be used for this purpose through the manipulation of a specific resource, ensuring a particularly useful communication when you want to steal sensitive information. We have two type of covert channels (Shrestha et al., 2015):

- *Covert Storage Channel*: it communicates information by modifying the data of a specific resource, which will then be read by the recipient.
- *Covert Timing Channel*: it works by altering the time required to perform a function or to use a resource, the recipient decodes the message by interpreting these time gap between packets.

In both cases it is necessary to access a shared resource by the process that sends the data and by the process that receives them, in order for communication to take place.

In (Marforio et al., 2012) authors report some of the most relevant covert channels, among which we have:

- *Single and Multiple Settings*: here the source application changes a smartphone system setting (i.e., volume, vibrations, screen) and the sink application then read the altered resource.
- *Type of Intents*: the data are sent via an intent, not by inserting them in the payload, but they are encoded in the intent type.
- *Threads Enumeration*: in this case the source application encodes the information for a certain number of threads, then the sink application reads data by monitoring the number of active threads.

- *UNIX Socket Discovery*: the source application uses two sockets, one for synchronization and one for communication, the application sink reads the data by checking the status of the latter socket.
- *Fee Space on Filesystem*: the information is sent by source application by writing and deleting data from the disk.
- *Timing Channel*: the source application performs high intensity activities to send bits of value 1, instead, for the transmission of bits of value 0, it does not perform any activity. The application sink continuously performs high-intensity activities and depending on the time it takes to complete them, is able to detect the message.
- *Processor Frequency*: it is an improvement of Timing Channel, where the source application have the same behaviour just described, instead the sink application checks the CPU frequency with queries and read the bit sent via the alterations caused by the source.

3 THE COLLUDING COVERT CHANNEL ATTACK DESIGN

In this work we have designed and developed a data-set composed by six source applications to obtain sensitive and private data and another one (i.e., the sink application) that receives it. In order to launch a colluding attack, the source applications have the necessary permissions to access the data of their interest, while the sink applications have access permissions to internet so they can transmit this data to the attacker.

3.1 Source Applications

We implement a hidden communication for the information exchange, based on the variation of the accelerometer data on Android devices. To encode the messages, the source applications exploit the vibration engine considered to modulate the sensor data able to detect device movements. Consequently, the sink application extracts the data contained into the variation of vibrations, as shown in Figure 1. This communication takes place during the night hours, when the device is less used, so as not to arouse suspicion in the unaware user.

Once the authorizations to read the data have been received, the source applications start a service to set up the encryption, the synchronization of the transmission and the sending of data. The required permissions are:

- *Foreground_Service*: they are used to ensure that the background services of the source applications are not subject to limitations due to excessive battery consumption. The Foreground_Services manifest their activity to the user through a notification that cannot be deleted unless the service is stopped and they continue their execution even if the user does not interact with the application.
  To start the Foreground_Service, the application uses the *startForegroundService()* method and it has five seconds to call *startForeground()* method, in order to show the notification before the service is stopped. This procedure needs the appropriate permission.
- *Vibrate*: it gives permission for access to the vibration motor, which guarantees the encoding of the data to be sent to the sink application by altering the sensor values.
- *Wake_Lock*: they keep the CPU active and prevent the screen from turning off. The device cannot enter sleep state as long as there is an active Wake_Lock, thus causing the battery to drain quickly. To take advantage of this permission, the *PowerManager* class is used.

In addition to the permissions listed above, which being normal permissions do not require the consent of the user, each application of the data-set has a fourth permission that must be explicitly given by the user and changes according to the information to be accessed (and which will then be sent to the sink application).

The developed applications are able to steal sensitive and private information, such as:

- *SMS*: using the READ_STATE permission, a malicious application has the ability to read the last SMS sent (or received), thus accessing the user's private conversations.
- *E-mail Address*: using the GET_ACCOUNTS permission, the application is able to retrieve all e-mail addresses stored in *AccountManager*.
- *Telephone Number*: using the READ_PHONE_STATE permission, the application can access to the telephone number and know the information about the network and the status of ongoing calls.
- *IMEI code*: it is a 15 digit numeric code that uniquely identifies the smartphone. It is useful for locking the device in case of theft and provides its location at a given time. Also in this case the permission used to obtain the IMEI code is READ_PHONE_STATE.

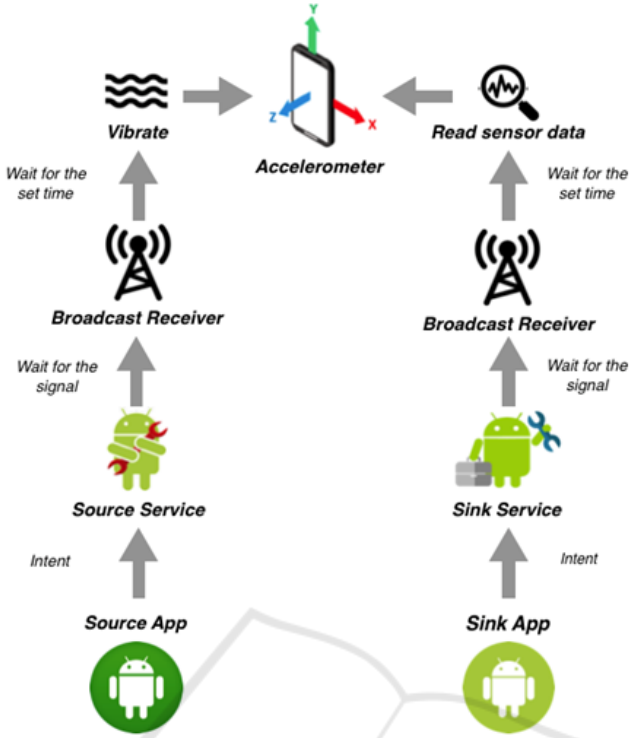

Figure 1: Workflow of the proposed colluding attack.

- *Contact List*: using the READ_CONTACTS permission, the malicious application is able to access to the user's contact list by obtaining for each contact: number and name. The attacker can thus use the contacts obtained to obtain others and launch new attacks.
- *Calendar Events*: using the READ_CALENDAR permission, the application can read information about the events contained into the calendar, such as title, description, position, date and time. The attacker can thus learn the user's habits or know where he will be on a certain day at a certain time.

The antimalware are not able to classify these applications as threats and the same goes for users, because they can only read data. It will be the sink application to help them to share the information through the network.

The six source applications are composed by the same components, the difference concerns the permissions to read data described above. We have three components: *MainActivity*, *SourceService* and *AlarmBroadcastReceiver*.

MainActivity requires the permissions for access to sensitive information, once obtained, by means of an intent is started *SourceService*, thanks to *startForegroundService()* method. The *SourceService* calls the *createNotification()* method, which is used to create the notification to be passed as a parameter to the *startForeground()* method, to avoid stopping the service.
The *SourceService* collects the target data of the attack through a different method for each of the six source applications (in some cases this function is performed by the *MainActivity*, which will then pass the information within the intent to start the service).
Using an *AlarmManager*, the time at which synchronization with the sink application takes place to start sending data is set. Then *AlarmManager* creates an alarm that sends an intent to the *AlarmBroadcastReceiver*, so that the service can transmit the data every day at the same time (Figure 2).

Before sending the data, the service must encode them in such a way as to create a vibration pattern to allow the accelerometer values to be changed. The information must be converted to binary, as there are two states of the vibration engine: 1 vibration, 0 no vibration. Extended ASCII code, an 8-bit coding system, was used for data encoding.

The *Vibrator* class is used to manage vibrations, which, using the *vibrate()* method, makes the smartphone vibrate in a certain way. An array of type long is passed to the method indicating how to turn vibration on and off. Each element of the array represents the time (in milliseconds) in which the device must vibrate or not. The first value indicates the time to wait before the vibration motor is activated; the

```
alarmManager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
calendar = Calendar.getInstance();
calendar.set(calendar.get(Calendar.YEAR),
        calendar.get(Calendar.MONTH),
        calendar.get(Calendar.DAY_OF_MONTH),
         hourOfDay: 3, minute: 0,  second: 0);

Intent secondIntent = new Intent( packageContext: this, AlarmBroadcastReceiver.class);
secondIntent.putExtra( name: "start", value: 1);
PendingIntent pendingIntent = PendingIntent.getBroadcast( context: this
        , requestCode: 0,secondIntent, flags: 0);
alarmManager.setRepeating(AlarmManager.RTC, calendar.getTimeInMillis(),
        AlarmManager.INTERVAL_DAY, pendingIntent);
```

Figure 2: Code snippet about the time setting to synchronize the two colluding apps.

second value indicates the time in which it must vibrate; the third value indicates the time in which it must deactivate and so on, alternating periods of vibrations and periods of ”rest”. The *createPattern()* method was implemented to create the pattern. With the *onSensorChanged()* method, contained in the *SensorEventListener* class implemented by the service, it is possible to check whether the user is using the device or not. This method is called when there is a new sensor event. If *onSensorChanged()* does not detect any movement of the device during the agreed time period, it calls *sendData()* after this time, with which the actual data transmission begins. The *sendData()* method invokes *createPattern()* and gives the created pattern to the *Vibrate* method, which receives the value ”-1” to avoid to repeat the vibration path just executed. Then is invoked *postDelayed*, a method that uses two parameters (Runnable and an Int value) to postpone *screenOn* method according to the duration of the vibration path.

3.2 Sink Application

The sink application, after starting the service, synchronizes with the source application and begins reading the accelerometer data, recording them and entering them into the network. The sink application requires two permissions:

- *Foreground_Service*: as described above, it creates the service for communication.
- *Internet*: it allows to use network sockets to connect to the internet, so as to send data to the attacker and terminate the attack.

Also the sink application is composed by three components: *MainActivity*, *SinkService* and *AlarmBroadcastReceiver*.

MainActivity and *AlarmBroadcastReceiver* perform the same functions described for the source application, except for requesting permissions, which is not needed for the sink application. The *SinkService* must call the *createNotification()* to avoid a shutdown and then set the alarm to synchronize its operation with that of the other applications.
Once it receives the signal from the *AlarmBroadcastReceiver*, it checks that the device is not using *onSensorChanged()*. This method assigns the acceletometer three float variables, according to the *x*, *y* and *z* coordinates, allowing us to read the changes in the sensor data caused by vibrations and capture the information sent. If the smartphone movement is detected, the communication stops, otherwise the service invokes *receiveData()*, that checks that the device is suspended. The communication then continues in the manner described above, until the source application communicates to the *SinkService* that it can terminate the data collection.

If the screen is off, the *postDelayed()* method is called by the handler to start *recData()* in order to read the data in another way: this method records the values related to the three of the accelerometer in a float-type array, repeating this process several times during the time it takes the source application service to send a single bit. Meanwhile, the first mode is performed, which consists of inserting bits with value 1 or value 0 in an int type array, based on the value assumed by one of the three sensor coordinates. Each time interval (equals to the milliseconds in which a single bit is sent by the *SourceService*) is restarted *receiveData()*. If the screen turns on, the *PowerManager* class, having detected the *Wake_Lock*, calls the method to save the float array in an internal application file and the one to decode the bits, extracting the information (Figure 3).

4 ANTIMALWARE ANALYSIS

To test the effectiveness of attack model we designed, the developed Android applications have been in-

```
Runnable receiveData = new Runnable() {
    @RequiresApi(api = Build.VERSION_CODES.P)
    @Override
    public void run() {
        if (!powerManager.isInteractive()) {
            handler.postDelayed(recData, delayMillis: VIB_TIME/INTERVAL);
            if ((zAxis > 9.9) || (zAxis < 9.7)) {
                arrayBin.add(1);
            } else arrayBin.add(0);
            handler.postDelayed(receiveData, VIB_TIME);
        } else {
            saveData(arrayFloat.toString());
            toASCII();
        }

    }
};

Runnable recData = new Runnable() {
    @Override
    public void run() {
        arrayFloat.add(xAxis);
        arrayFloat.add(yAxis);
        arrayFloat.add(zAxis);
    }
};
```

Figure 3: Code snippet related to the accelerometer data collection.

stalled on three different real-world devices: Samsung Galaxy S8 (released in 2017, Android version: Android 7.0 Nougat), Huawei P10 Plus (released in 2017, Android version: Android 7.0 Nougat), Oppo Reno2 (released in 2019, Android version: Android 9.0 Pie).

On all the considered Android device models the attacks were correctly perpetrated. During the experimental analysis, the applications always managed to synchronize, sending and receiving data on schedule. Tests were carried out on the ability to be able to pick up data, encode them without errors in the vibration pattern, and the actual ability of the channel to carry data was tested, so that the sink application can receive the correct message. The channel has been tested by sending messages of various lengths, based on the type of information contained.

Here there is an example of a message exchanged between the colluding apps based on the e-mail, along with its binary representation:

attacker@collusion.com

01100001 01110100 01110100 01100001 01100011 01101011 01100101 01110010 01000000 01100011 01101111 01101100 01101100 01110101 01110011 01101001 01101111 01101110 00101110 01100011 01101111 01101101

Each bit is transmitted with an interval of 400 milliseconds, making the vibration motor follow the following pattern:

0, 800, 1600, 400, 400, 1200, 400, 400, 1200, 1200, 400, 400, 1200, 800, 1600, 400, 400, 800, 1200, 800, 400, 800, 400, 400, 400, 800, 400, 800, 800, 400, 400, 400, 400, 1200, 800, 400, 800, 400, 2800, 800, 1200, 800, 400, 800, 400, 1600, 400, 800, 400, 800, 1200, 800, 400, 800, 1200, 1200, 400, 400, 400, 400, 400, 1200, 800, 800, 400, 800, 400, 400, 800, 400, 400, 800, 400, 1600, 400, 800, 400, 1200, 1200, 400, 400, 1200, 800, 800, 1200, 800, 400, 800, 400, 1600, 400, 800, 400, 800, 400, 400

The part highlighted in red in the pattern is the one corresponding to the first byte of the e-mail conversion to binary. When there are consecutive bits of the same type, the 400 milliseconds interval is added by their number (i.e., 011100 will be [0, 1200, 800]). The 400 milliseconds interval allowed to eliminate the error rate, otherwise the sink application would not be able to separate the bits correctly. It took 70.4 seconds for the message to be sent (with a transmission rate equal to 2.5 bps).

About the data extraction carried out on the device, we had problems choosing the coordinate from which to read the vibration: x, y, z or a com-

Table 1: VirusTotal classification results.

Application Name	Trusted For N° Antimalware	Not Trusted For N° Antimalware	FP
SinkApp.apk	62/62	0/62	0
SMSSourceApp.apk	61/62	1/62	1
EmailSourceApp.apk	63/63	0/63	0
TelSourceApp.apk	62/63	1/63	1
IMEISourceApp.apk	62/62	0/62	0
ContactsSourceApp.apk	62/62	0/62	0
CalendarSourceApp.apk	63/63	0/63	0

bination of them. Acceleration along the x axis and acceleration along the z axis were the best figures for Samsung and Huawei. For the Oppo it was only possible to obtain information on the x axis.

The applications developed have been checked by VirusTotal[3], an online antivirus service owned by Google, which scans files and URLs to detect any malware. The analysis takes place with more than 60 scanning engines and can also be used to find "false positives" (i.e., files that are not dangerous but recognized as such).

Once scanned, applications were not classified as malware. The SMS reader and the Telephone Number applications have been classified as generic threat by the TrustLook antimalware, as shown in the Table 1, due to the fact that they have permission to access messages and contact numbers, actions not allowed by the antimalware. For this reason, a test was performed, by submitting to VirusTotal an application with an empty activity and the READ_SMS permission (contained in both applications), and it was also classified as generic threat by the TrustLook antimalware.

5 RELATED WORK

Covert channels allow an hidden communication to launch a colluding attack, in this regard in (Wang et al., 2020) the authors have developed Multichannel Communication System (MSYM), a mechanism for the transmission of sensitive data in a mobile environment. Its operation is based on the use of the Android VpnService interface, allowing the interception of the data sent and split it in different parts that will be disordered and encrypted across multiple transmission channels.

The authors in (Al-Haiqi et al., 2014) have decided to exploit the accelerometer as a covert channel and have developed a source and a sink application in Android to demonstrate that through this sensor it is possible to pass encrypted information between applications. Applications cannot directly generate or manipulate sensor data to transmit information, they can only read sensor signals to obtain information about the environment and user actions.

MAGNETO (Guri, 2020) is a secret communication proposal on air-gap systems and nearby smartphones based on the magnetic fields generated by the CPU. This attack uses the magnetometer, which is the magnetic sensor inside the devices used for orientation and positioning, to steal data from computers. The generation of magnetic signals occurs as the CPU workload changes.

The researchers in (Denney et al., 2018) propose a method to create a covert channel starting from the notifications shown on an Android device when we receive e-mails or SMS messages. This covert channel uses Android Wearable notifications to send data through applications sited on the same device or on different devices. There are two applications: the first one creates the notification that will be used as covert channel, the second one reads the notification and determines the hidden message.

6 CONCLUSION AND FUTURE WORK

The large amount of resources present in mobile devices makes it possible to implement various types of covert channels to initiate a hidden communication of sensitive information. One of these is to use the accelerometer data, modified through the vibration motor according to certain coding patterns.

The proposed covert channel proved capable of transmitting data effectively and in a silent way. Such a channel is particularly difficult to detect, as it does not establish any explicit communication between colluding applications. This made it possible for applications to launch the attack, without being classified as threats.

As future work we plan to extend the communication to other sensors of the device, so as to create new

[3]https://www.virustotal.com/

covert channels. Moreover, we will study techniques aimed to detect this kind of communication.

As a matter of fact, we plan to apply formal methods for implementing an approach for identifying these communications, so as to demonstrate how it is possible to counter them. As a matter of fact, in literature formal methods already demonstrated their ability to detect malicious communication between Android applications (Iadarola et al., 2020; Casolare et al., 2020b).

ACKNOWLEDGEMENTS

This work has been partially supported by MIUR - SecureOpenNets, EU SPARTA, CyberSANE and E-CORRIDOR projects.

REFERENCES

Al-Haiqi, A., Ismail, M., and Nordin, R. (2014). A new sensors-based covert channel on android. *The Scientific World Journal*, 2014.

Canfora, G., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2018). Leila: formal tool for identifying mobile malicious behaviour. *IEEE Transactions on Software Engineering*, 45(12):1230–1252.

Casolare, R., Martinelli, F., Mercaldo, F., and Santone, A. (2020a). Android collusion: Detecting malicious applications inter-communication through sharedpreferences. *Information*, 11(6):304.

Casolare, R., Martinelli, F., Mercaldo, F., and Santone, A. (2020b). Malicious collusion detection in mobile environment by means of model checking. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6. IEEE.

Cimino, M. G., De Francesco, N., Mercaldo, F., Santone, A., and Vaglini, G. (2020). Model checking for malicious family detection and phylogenetic analysis in mobile environment. *Computers & Security*, 90:101691.

Cimitile, A., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2018). Talos: no more ransomware victims with formal methods. *International Journal of Information Security*, 17(6):719–738.

Denney, K., Uluagac, A. S., Aksu, H., and Akkaya, K. (2018). An android-based covert channel framework on wearables using status bar notifications. In *Versatile Cybersecurity*, pages 1–17. Springer.

Enck, W. (2011). Defending users against smartphone apps: Techniques and future directions. In *International Conference on Information Systems Security*, pages 49–70. Springer.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29.

Guri, M. (2020). Magneto: Covert channel between air-gapped systems and nearby smartphones via cpu-generated magnetic fields. *Future Generation Computer Systems*.

Iadarola, G., Martinelli, F., Mercaldo, F., and Santone, A. (2020). Call graph and model checking for fine-grained android malicious behaviour detection. *Applied Sciences*, 10(22):7975.

Mahboubi, A., Camtepe, S., and Morarji, H. (2017). A study on formal methods to generalize heterogeneous mobile malware propagation and their impacts. *IEEE Access*, 5:27740–27756.

Marforio, C., Ritzdorf, H., Francillon, A., and Capkun, S. (2012). Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016a). Hey malware, i can find you! In *2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 261–262. IEEE.

Mercaldo, F., Visaggio, C. A., Canfora, G., and Cimitile, A. (2016b). Mobile malware detection in the real world. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 744–746. IEEE.

Nguyen, T., Mcdonald, J., Glisson, W., and Andel, T. (2020). Detecting repackaged android applications using perceptual hashing. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*.

Shrestha, P. L., Hempel, M., Rezaei, F., and Sharif, H. (2015). A support vector machine-based framework for detection of covert timing channels. *IEEE Transactions on Dependable and Secure Computing*, 13(2):274–283.

Wang, W., Tian, D., Meng, W., Jia, X., Zhao, R., and Ma, R. (2020). Msym: A multichannel communication system for android devices. *Computer Networks*, 168:107024.