Malicious Collusion Detection in Mobile Environment by means of Model Checking

Rosangela Casolare*, Fabio Martinelli†, Francesco Mercaldo†*, Antonella Santone*
*Department of Biosciences and Territory, University of Molise, Pesche (IS), Italy
{rosangela.casolare, francesco.mercaldo, antonella.santone}@unimol.it
†Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy
{fabio.martinelli, francesco.mercaldo}@iit.cnr.it

Abstract—**Everyday born a new cyberattack and among these an emerging attack is represent by the so-called colluding. The application collusion attack is a new form of threat that is becoming widespread in mobile environment, especially in Android platform. This technique requires that two or more apps cooperate in some way with the aim to perform a malicious action that they are unable to perform independently. Detecting colluding apps is challenging problem, because currently there are no effective tools due to the search space of all possible combination of apps. In this paper we present a method exploiting model checking technique with the aim to detect a collusion attack between two applications. The method uses a heuristic function able to reduce the number of the analyzed apps and to localize the collusion attack. This heuristic function is based on the study of execution flow of an application, to identify the execution flow and verify it. The proposed algorithm verify if there is a flow of sensitive data that ends up in a shared resource and if this happens the app could be marked as potentially collusive, otherwise it is possible to exclude the app from the analysis, in order to reduce the number of apps to be analyzed. Experimental results on a data-set of Android applications show promising performances in colluding mobile app detection.**

Index Terms—**colluding, model checking, formal methods, Android, security**

I. Introduction

Everyday people use devices such as smartphones and tablets without thinking about problems related to their use, because smartphones handle a great deal of personal information (e.g. photos, finances, messages) that can be stolen. In fact Android platform being the most popular mobile operating system, is also the most attacked by cybercriminals. The large popularity of Android combined with its open nature made this operating system a primary target of attackers able to develop more and more malicious apps at an industrial scale [1]–[3].

From the defensive side, commercial and free antimalware software solutions are not able to correctly identify new threats, because the malicious payload is detectable only once its signature is stored in their repository.

In this scenario, malicious writers have developed a new trend, the so-called colluding attack. It basically consists in a malicious action split into several apps, so current antimalware that analyze a single application to discover harmful actions as a single entity, are not able to identify the malicious payload [4]. These applications communicate with each other when the user performs a specific action or when a specific system event occurs and not make suspicious the antimalware, because they require only the minimum permissions needed for play their role in the attack [5]: the first app might read sensitive data and transmits it to the second one, which transmits they to the outside. In this way the first application requires only the permissions to read the data, instead the second one requires the permissions to use the Internet connection [5]. If two apps that collude are individually analyzed, for the antimalware will not be possible to find errors or threats, since the damage will be caused by their collusion [6].

In Android the applications not always are independent to each other, they may communicate through the Inter-Component Communication (ICC). This mechanism reduces the developers' burden and promote functionality reuse, allowing an inter-applications collaboration thanks to information exchange between components that could be of the same application or different applications [7], but it can be also misused by malicious applications to attack users privacy [8].

Starting from these considerations, in this paper we present a tool implementing a methodology able to detect colluding Android applications based on model checking technique [9] and on a new heuristic function that aims to reduce the number of the analyzed apps. To define the function has been used the μ-calculus temporal logic and it is based on model checking. In particular in this paper we focus on *string* resources shared exploiting Android *SharedPreferences*.

The paper is organized as follows. Next section introduces the proposed method for detecting colluding attacks in mobile environment, Section III presents the experimental analysis we performed to demonstrate the effectiveness of the proposed method and, finally, in last section conclusion and future research lines are drawn.

II. The Method

To detect and verify colluding Android apps, we have build the methodology starting from the binary code. The choice is fell on the application Bytecode since this type of code is always reachable as opposed to the source code that is not always possible to achieve due to the obfuscation of the same. The first step consists to define the formal model that allows to create a model of the Android application. In this way you have a general model with which is possible check every type of property on the system.

A. Formal Model Creation

The formal model is created using the Calculus of Communicating Systems (CCS) process that simulates the behaviour of an application, starting from the bytecode. An Android app, also called .apk (i.e., Android Package) is a variant of the well-known .jar archive file. This type of file contains the executable code for the Dalvik Virtual Machine (i.e., the .dex file), the re-source folder (i.e., icons, images, sounds) and the Manifest file.

Starting from the .apk we can obtain Bytecode instructions. The process to transform an apk file into a Javabytecode is composed by following steps:

- generation of the .jar file from the .apk file with a tool called dex2jar[1];
- extraction of the java class file and the directories from the .jar file, using the Java Archive Tool utility[2];
- generation of the Android application Bytecode invoking the BCEL (Byte Code Engineering Library[3]).

After obtained the Java Bytecode, is used an inference algorithm developed by the authors in [10]. This algorithm generates a CCS process for each Java Bytecode instruction. The CCS process encodes the instructions, the actions that it makes, represent the opcodes (i.e., the control-flow between two or more instructions).

B. Heuristic function: PUT and GET

The detection of colluding apps is very difficult, because there are not effective tools able to search all possible combination of apps. There is a large number of applications in official and unofficial stores and this is the reason for the exponential growth of analysis costs: if we have n apps, analyzing all the possible pairs we obtain n^2 tests have to be performed, considering all the possible triples of colluding apps instead are required n^3 tests and so on (Figure 1).

A method is needed to reduce the search for collusion candidates and to identify which groups of apps should be considered together for collusion.

In this paper are presented two different heuristic functions: the first one is based on the uses of the *SharedPreferences* and monitors every possible code path for each read/write of a *string* shared resource. It makes a division of the apps based on the different use of the shared resources.

To show how *SharedPreferences* are working, let us consider the Android code snippet below shown: it represents an example of *SharedPreferences* invocation, in particular the code snippet retrieves by invoking the *getString* methods a string value from the *SharedPreferences* (stored in the *value1* variable).

```
SharedPreferences sharedPreferences = this.
    getSharedPreferences("SharedPreferences",
    Context.MODE_WORLD_WRITEABLE);
String value1 = sharedPreferences.getString("value1"
    ,"defaultValue");
```

[1]https://sourceforge.net/projects/dex2jar/
[2]https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jar.html
[3]https://commons.apache.org/proper/commons-bcel/

The second Android code snippet presents the *SharedPreferences* writing invocations.

```
SharedPreferences sharedPreferences= this.
    getSharedPreference("SharedPreferences", Context
    .MODE_WORLD_WRITEABLE);

SharedPreferences.Editor editor = sharedPreferences.
    edit();
editor.putString("value1", "information");
```

In particular the information in the *SharedPreferences* are stored using the *putString* methods.

As discussed in literature [6], it is very easy for two applications to share (sensitive) data by only knowing the name of the *SharedPreferences* (in the code snippets *SharedPreferences*).

Usually an application can execute two different type of operation on a shared resource (i.e., *get* and *put* operations).

We can define these actions after the study of the application behaviour.

We resort to the μ-calculus logic to encode these actions:

- an app can perform a ”put” on a shared resource, in this case the formula (Table I - $Formula1$) is true if the process is able to perform the following sequence of actions: $InvokegetSharedPreferences$, $invokeedit$, $invokeputString$, $invokecommit$;
- an app can perform a ”get” on a shared resource, in this case the formula (Table I - $Formula2$) is true if the process is able to perform the following sequence of actions: $InvokegetSharedPreferences$, $invokegetString$.

The developed methodology uses the heuristic function to search a pair of colluding apps in a short time. With the heuristic function is possible to create two different sets of applications which will be analyzed: the first one containing the applications that verify the put property (Figure 2), the second one containing the applications that verify the get property (Figure 3).

Furthermore the heuristic function allows to reduce the computing cost since it is based on checking a temporal logic formula in the CCS processes representing the applications.

The second heuristic function works after the put property verification and is useful to further reduce the search space of the applications that collude based on the execution flow. To verify if there is flow, the tool use FlowDroid.

We need also a system model and system property to apply the model verification technique. To create the system model has been developed an algorithm that start from FlowDroid, it takes in input an application at a time returning an xml file containing the reconstruction of data flow and the description of all sources with the respective sinks. In this way will be created two lists: one for the sources and one for the sinks. These lists are then inserted into hashmap where the source represent the key and the array of sink represent the respective value. To create the model, the flow is modeled as a graph composed by an array of roots and an array of leaves: the sources array is scanned and if the source element is present also in the sinks array, it is deleted from both arrays.

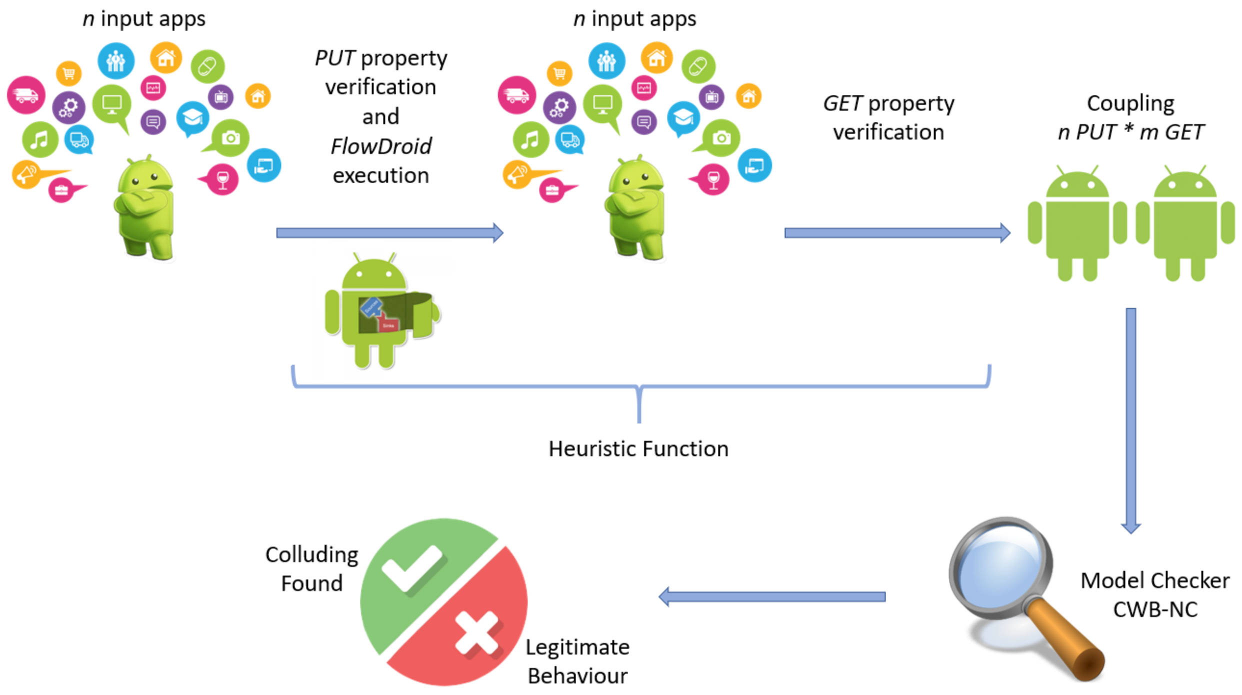
n input apps
PUT property verification and *FlowDroid* execution
n input apps
GET property verification
Coupling *n PUT* * *m GET*
Heuristic Function
Model Checker CWB-NC
Colluding Found
Legitimate Behaviour

Fig. 1. The proposed methodology.

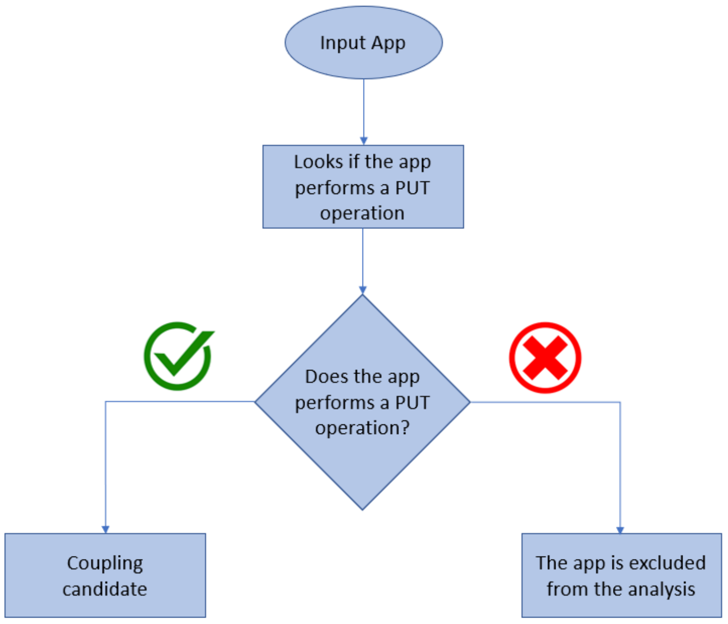
Input App
Looks if the app performs a PUT operation
Does the app performs a PUT operation?
Coupling candidate
The app is excluded from the analysis

Fig. 2. First heuristic flowchart about PUT property.

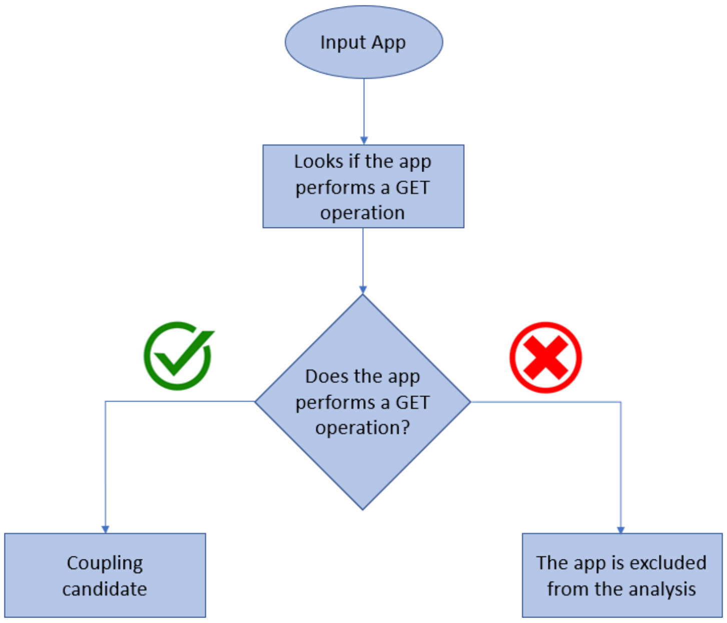
Input App
Looks if the app performs a GET operation
Does the app performs a GET operation?
Coupling candidate
The app is excluded from the analysis

Fig. 3. First heuristic flowchart about GET property.

TABLE I
GET AND PUT FORMULAE RELATED TO THE HEURISTIC FUNCTION

$Formula1$		
φ_{PUT}	$=$	$\mu X.\,\langle invokegetSharedPreferences\rangle\ \varphi_{PUT_1} \vee \langle - invokegetSharedPreferences\rangle\ X$
φ_{PUT_1}	$=$	$\mu X.\,\langle invokeedit\rangle\ \varphi_{PUT_2} \vee \langle - invokeedit\rangle\ X$
φ_{PUT_2}	$=$	$\mu X.\,\langle invokeputString\rangle\ \varphi_{PUT_3} \vee \langle - invokeputString\rangle\ X$
φ_{PUT_3}	$=$	$\mu X.\,\langle invokecommit\rangle\ \mathtt{tt} \vee \langle - invokecommit\rangle\ X$
$Formula2$		
φ_{GET}	$=$	$\mu X.\,\langle invokegetSharedPreferences\rangle\ \varphi_{GET_1} \vee \langle - invokegetSharedPreferences\rangle\ X$
φ_{GET_1}	$=$	$\mu X.\,\langle invokegetString\rangle\ \mathtt{tt} \vee \langle - invokegetString\rangle\ X$

About our work is been developed a recursive algorithm called $CodeSnippet1$, it selects the first root from the roots array, enters in the hashmap (using the root that represents a key) and then finds if this root has sons. If the root has a son, in the CCS file model we go from the root to the son and the recursive algorithm is invoked until all the roots have been selected. In the end, you get the CCS file ready for checking the model.

C. Coupler definition and formal verification

Using the above explained heuristic functions, we obtain a set of app pairs for the *SharedPreferences*, following indicated as S_{sp}.

For each $(p, q) \in S_{sp}$ we define the following CCS process:

$$Proc_{pq} = (p\ |C|\ q)\backslash L$$

where:

- p and q are the CCS representation of the apps that potentially may colluding, since obtained by the heuristic functions.
- C is the coupler definition process that has the aim to identify whether p and q collude.
- L is the set of communication actions. $L = \{Preferences\_NAME,\ invokeSharedPreferences,\ resource\_ID,\ invokeputString,\ invokegetString\}$.

The last step consists in the application of a formal verification environment including a model checker. The CCS process $Proc_{pq}$ is checked for each $(p, q) \in S_{sp}$. When the result of the CWB-NC (the model checker tool used) is true, CCS process $Proc_{pq}$ satisfies a μ-calculus formula encoding the colluding notion, it means that our method considers two apps p and q under analysis colluding, false otherwise.

III. The Experiment

In this section we present the experimental analysis, aimed to demonstrate the effectiveness of the proposed method for detecting colluding applications.

A. The data-set

To test our methodology the data-set has been generated using different sources:

- We have used a data-set generated with Application Collusion Engine (ACE) [11], a system to automatically generate combinations of colluding and non-colluding Android apps to evaluate different collusion detection and protection methods. For the experiment we have generated 80 couples of colluding apps (160 different applications) communicating through *SharedPreferences* and 160 couples of colluding apps (320 different applications) not using this type of communication. Therefore, is been considered a data-set composed by 480 colluding apps, where only 160 of they show the malicious action with the *SharedPreferences*;
- the second data-set considered is DroidBench 2.0[4], an open data-set to evaluate the effectiveness of taint-analysis tools specifically for Android applications. In this data-set there are no apps that collude, only some use *SharedPreferences*;
- we have also a colluding data-set created by Swansea University containing 14 applications, where there is only a pair of different apps that colludes;
- have been taken a set of "trusted" apps from Google Play Store, divides by their size (from 500kb to 700kb) and a set composed by "trusted" apps taken from the web[5].

B. Results

To measure the performance of the methodology we have used the following metrics:

$$Precision\ (PR) = \frac{TP}{TP+FP}$$

$$Recall\ (RC) = \frac{TP}{TP+FN}$$

$$F-measure\ (Fm) = \frac{2*(PR*RC)}{PR+RC}$$

$$Accuracy\ (AC) = \frac{TP+TN}{TP+FN+TN+FP}$$

where TP (True Positive) represents the number of colluding apps couples identify correctly with the tool, TN (True Negative) represents the number of not colluding apps couples identify correctly with the tool. FP (False Positive) represents the number of not colluding couples that are identified from the tool as colluding couples and FN (False Negative) represents the number of colluding couples that are identified from the tool as not colluding couples.

In Table II are showed the obtained results. We obtain an interesting rate of precision and recall, furthermore during the colluding identification process we have an accuracy ranging between 0.98 and 1.

The developed methodology is able to correctly identify all the 80 couples of colluding applications showing the presence of malicious behaviour using the *SharedPreferences*.

With the just described heuristic function, it is possible to consider $6,400$ possible couples of different applications. Without using the heuristic function, the theoretical number of actual couples of apps is equal to the binomial coefficient, in fact there are $\binom{320}{2}$ possibility to choose a subset of 2 elements from a set containing 320 elements.

Also about the analysis of trusted applications the method achieves good results, giving for all trusted data-sets the indication of not malicious apps.

IV. Related Work

Nowadays the detection techniques generally used by commercial and free antimalware software, focus the attention on the analysis of one sample at a time, without consider the communication channel exploited by two or more applications to perpetrate a malicious behaviour. For this reason, the research community is working to develop new methods for the identification of malicious behavior linked to the use of communication channels. These channels can be used from different applications to communicate with another application and this communication could be malicious, such as for the colluding technique.

For instance, DroidSafe [12] is a tool that analyze static information flow to report possible leaks of sensitive data in Android applications. To develop DroidSafe has been used Soot Java Analysis Framework and it works by analyzing one app at a time, but the analysis of a single app does not permit the detection of colluding attacks that are caused by two or more apps.

TaintDroid [1] is an Android operating system extension and tracks the flow of privacy sensitive data through third-party applications. It assumes that downloaded third-party applications are not trusted and monitors in realtime how these applications access and manipulate users' personal data.

IccTA [13] uses a static taint analysis technique to find privacy leaks, e.g. paths from sensitive data (called sources) to statements sending the data outside the app or device (called sinks). This path may be within a single component or across multiple components. The researchers have developed about 22 apps containing ICC-based privacy leaks to verify this approach.

Epicc [14] identifies a specification for every ICC source and sink. It works on the location of the ICC entry/exit point, the ICC Intent action, data type and category, the ICC Intent key/value types and the target component name. Epicc infers all possible ICC values where they are not fixed, thereby building a complete specification of the possible ways ICC can be used. All the specifications are then recorded in a database as flows detected by matching compatible specifications.

Another kind of colluding attack is about the covert channel. Usually mobile communication systems use a single channel to

[4]https://github.com/secure-software-engineering/DroidBench
[5]http://www.freewarelovers.com/android/apps

TABLE II
COLLUDING EVALUATION

Apps	Theoretical Couples	Effective Couples	Colluding Couples	TP	TN	FP	FN	PR	RC	Fm	AC
773	298378	6668	81	80	298296	1	1	0.98	0.98	0.98	0.99

transmit the data, but the authors in [15] propose a multichannel communication mechanism for mobile devices to secure sensitive data transfer, called Multichannel Communication System (MSYM). They use the VpnService interface provided by Android platform to intercept the network data sent and then split it into different parts that will be disordered and encrypted via multiple transmission channels.

A new type of covert channel is represented by the accelerometer sensor, able to generate signals that reflect the users' motions. As a matter of fact, a malicious application can read this data, but if the device vibration motor is used properly, is possible to encode stolen data, so an application can produce effect on acceleration data to be received and decoded by a second application. About that in [16] are been implemented two Android apps (source and sink) and used three different smartphones to verify this type of communication.

The approaches just discussed not consider the problem of a large number of Android applications that need to be analyzed to find colluding. Furthermore, they analyze only an app at time without the possibility to localize the malicious payload perpetrating the colluding attack, we instead present in this paper an approach able to analyze more applications at time.

V. CONCLUSION AND FUTURE WORK

Collusion is an emerging type of attack afflicting mobile environment, it is based on the communication between two or more malicious applications.

The attack is performed by the collaboration of the applications: a single app is not able to performs the malicious action by it-self, so the current antimalware cannot identify these apps as dangerous whether analyzed separately. About that, in this paper is showed a model checking-based method to detect this threat in Android environment.

We presented a heuristic function to reduce the number of apps to analyze, using the μ-calculus temporal logic. Different real-world data-sets containing collusive applications are considered, obtaining an accuracy equal to 0.99.

As future work, we plan to improve the method to analyze others app components extending the applications' dataset (for istance, *Broadcast Receivers*) currently exploited to circumvent the current Android security model. Moreover, We are working to permit to the proposed method to detect collusion attacks requiring more thank two applications in fact, on of the weakness of the proposed method is that is able to test only two apps at a time for now. Furthermore we are thinking a method to sanitize the infected apps (i.e., by removing the actions involved in the colluding action).

ACKNOWLEDGMENTS

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

REFERENCES

[1] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[2] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *USENIX security symposium*, vol. 31. San Francisco, CA;, 2011, p. 3.

[3] W. Enck, "Defending users against smartphone apps: Techniques and future directions," in *International Conference on Information Systems Security*. Springer, 2011, pp. 49–70.

[4] A. M. Memon and A. Anwar, "Colluding apps: Tomorrow's mobile malware threat," *IEEE Security & Privacy*, vol. 13, no. 6, pp. 77–81, 2015.

[5] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 51–60.

[6] R. Casolare, F. Martinelli, F. Mercaldo, and A. Santone, "A model checking based proposal for mobile colluding attack detection," in *IEEE Big Data*, 2020.

[7] K. Xu, Y. Li, and R. H. Deng, "Iccdetector: Icc-based malware detection on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[9] M. G. Cimino, N. De Francesco, F. Mercaldo, A. Santone, and G. Vaglini, "Model checking for malicious family detection and phylogenetic analysis in mobile environment," *Computers & Security*, p. 101691, 2019.

[10] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, "Ransomware steals your phone. formal methods rescue it," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2016, pp. 212–221.

[11] J. Blasco and T. M. Chen, "Automated generation of colluding apps for experimental research," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 2, pp. 127–138, 2018.

[12] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[13] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

[14] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 543–558.

[15] W. Wang, D. Tian, W. Meng, X. Jia, R. Zhao, and R. Ma, "Msym: A multichannel communication system for android devices," *Computer Networks*, vol. 168, p. 107024, 2020.

[16] A. Al-Haiqi, M. Ismail, and R. Nordin, "A new sensors-based covert channel on android," *The Scientific World Journal*, vol. 2014, 2014.